Version: August 8, 2002

SSC

User's guide

A stochastic simulation library in C

Pierre L'Ecuyer[1]

Département d'informatique et de recherche opérationnelle
Université de Montréal

Abstract. SSC is a software library implemented in ANSI C, offering basic tools for random number generation and stochastic discrete event simulation. It provides facilities for Monte Carlo and quasi-Monte Carlo methods, clock and event list management, continuous simulation (where the evolution is via differential equations), and some statistical analysis. The companion library SSJ provides an additional layer of software tools on top of SSC for simulation programming in Java.

[1]Francis Picard, Jean-Sébastien Senécal, and Richard Simard have contributed to the development of SSC.

Contents

1 An overview of SSC **2**

APPENDICES **19**

A. Functional definition of SSC **19**

Rand 20

Mrg32k3 24

Qrand 27

Qlcg 30

Qtaus 31

Fdist 32

Rand1 34

Rand2 36

Randlib 41

Sim 43

Event 44

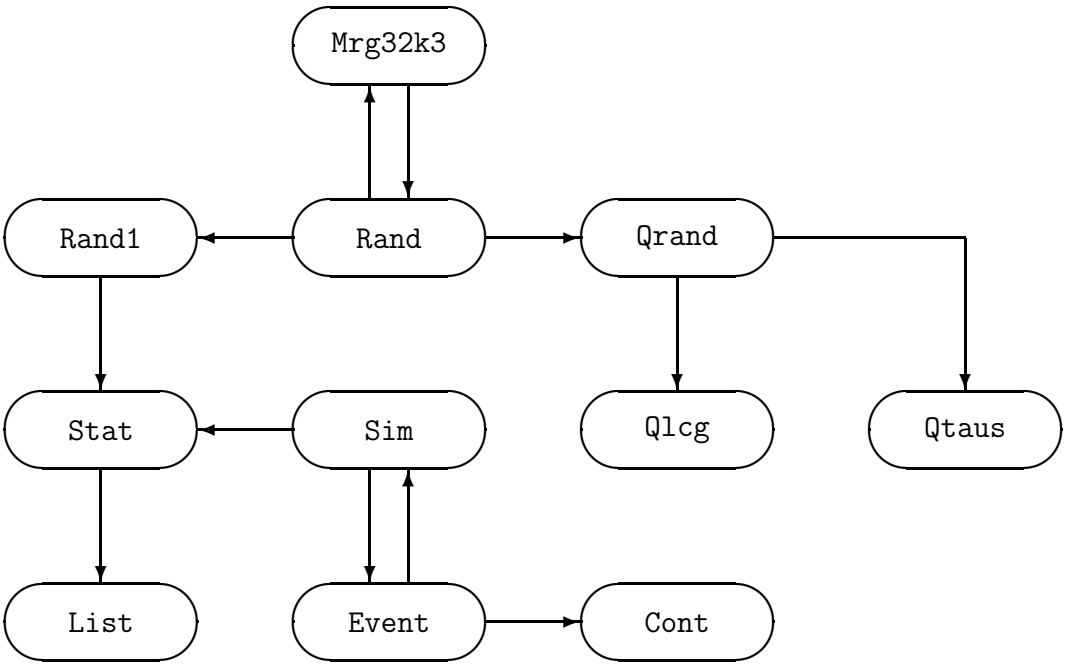
Cont 47

Stat 49

List 53

1 An overview of SSC

We now describe the organization of SSC as a hierarchy of *modules*. Each module has a header file (`.h`) and an implementation file (`.c`), and implements a set of related types and procedures. A detailed description of all the modules can be found in Appendix A. The following diagram illustrates the hierarchical structure between the different modules. [2] If there is a path going from module A to module B by following the arrows, this indicates that module B uses module A in its definition or in its implementation. These modules use the standard ANSI-C library, and many of them also use the home-made library MYLIB-C, implemented in ANSI-C, which contains additional utilities and comes with the SSC package. The users can of course add more modules if needed.



[2] From Pierre: This diagram is for the 1999 version.

Here is a brief description of the modules available.

- `Rand` contains the basic structure for creating random number generators.
- `Mrg32k3`, `Combtaus113`, ... are implementations of specific random number generators, with splitting facilities.
- `Qrand` defines the basic structures to manipulate highly-uniform point sets (HUPS) for quasi-Monte Carlo methods.
- `Qlcg`, `Qtaus`, ... implement specific HUPS.
- `Rand1, Rand2, Rand3` provide procedures for generating random variates from a variety of probability distributions. The modules `Rand2` and `Rand3` are interfaces to the packages `C--RAND` of Kremer et Stadlober [29, 43] and `Randlib` of Brown et al. [11].
- `Sim` manages the simulation clock and provides procedures to initialize, start, and stop the simulation.
- `Event` manages the event list in the simulation. It offers various tools to schedule or cancel events.
- `Cont` provides basic tools for continuous simulation, where certain variables evolve according to differential equations.
- `Stat` provides facilities for the collection of statistics.
- `List` implements doubly linked lists of objects of arbitrary types, and a set of list management facilities. The lists are managed without paying attention to the types of the objects that they contain. In fact, the objects are viewed by the software as having the type `void *`, which is compatible with any pointer type. The lists are thus lists of pointers, and can contain practically any kinds of objects. They can be used to implement, e.g., waiting lists. Facilities are also provided to collect statistics on those lists.

1.1 Example Q: A single-server queue

To illustrate how a discrete-event simulation runs, we take a very simple example and work it out. This model is a *single-server queue* and all along the document, it will be referred to as Example Q. It is defined as follows. Customers arrive randomly to a single server. They are served one by one and the order of service is first in, first out (FIFO). Let S_i denote the service time of the ith arriving customer and A_i be the time between the arrivals of customers i and $i+1$. The S_i's and A_i's are mutually independent random variables, with respective distributions G and F. At time 0, the system is empty. The first customer arrives at time A_0, leaves at time $A_0 + S_1$, and so on.

This simplistic model is hardly to be taken seriously as representative of real-life systems that need to be simulated. Its purpose is to give the flavor of how simulation works. For more concreteness, one can view the server as either a clerk, a machine, a computer, or a dock, and the customers as people, parts, jobs, or boats. So, the system could be boats coming to a dock to be loaded with, say, iron ore.

To simulate such a system, one needs to generate the random variables $A_0, A_1, \ldots$ and $S_1, S_2, \ldots$ (not necessarily in that order) and, from that, to retrace the sequence of events e_i together with their epochs t_i. In typical discrete-event simulations, this is implemented using a list of event notices, also called the *event list*. Each such *event notice* indicates an event that is already scheduled to occur in the future, and gives its scheduled time of occurence, with possibly some additional parameters. Event notices are ordered in the list by increasing (scheduled) order of occurence. During the execution of the simulation, event notices are added to the list dynamically (event scheduling) and the simulation program repeatedly removes the first event notice from the list, sets the simulation clock to the epoch of the corresponding event, and executes that event, until either a particular event stops the simulation or the event list gets empty. In some cases, event notices may also be removed before ever reaching the head of the event list (event cancellation).

Events are typically classified into several event *types*, and each event type may correspond to a piece of code to be executed by the program. The role of that code is to update the state of the system appropriately, given that an event of that type occurs. This may include, in particular, generating certain random variables and scheduling other (future) events.

Suppose we want to simulate the single-server queue for T units of simulated time, where T is fixed. We consider three types of events here: *arrival*, *departure*, and *end-of-simulation*. An event of the latter type occurs only once, when time T is reached. The corresponding procedure prints a report on whatever statistics we are interested in, and stops the simulation. Now, suppose that our program has a special data type called *customer* and maintains a list of the customers who are waiting in the queue (called the *waiting list*), together with their arrival times, as well as a list of customers who are being served (called the *server's list*), with the times at which their services began. This may seem a bit more than necessary, in particular because the latter list will never contain more than one customer, but we prefer this data structure because it will also work if the queueing model has more than one server,

and in other more general situations. The *arrival* and *departure* events can be implemented by procedures which we outline as follows. (Indentation delimits the scope of the IF and ELSE statements.)

```
PROCEDURE arrival;
    IF the server's list is not empty THEN
        Insert this new customer at the end of the waiting list;
    ELSE
        Insert this new customer in the server's list;
        Generate a random variable S from distribution G;
        Schedule a departure in S time units;
    Update the statistics as needed;
    Generate a random variable A from distribution F;
    Schedule an arrival in A time units.
```

```
PROCEDURE departure;
    Remove the departing customer from the server's list;
    IF the waiting list is not empty THEN
        Remove the first customer from the waiting list;
        Insert it in the server's list;
        Generate a random variable S from distribution G;
        Schedule a departure in S time units;
    Update the statistics as needed.
```

At the beginning of the simulation, one must initialize the lists and the appropriate statistical accumulators, schedule the end-of-simulation event at time T, generate A_0 from distribution F, schedule an arrival at time A_0, and start the simulation monitor, which will remove the first event in the event list, execute it, and so on.

The question of which statistics should be collected depends on what we want to estimate. For example, suppose that we are interested in (a) the average waiting time in the queue per customer for those customers who have started their service, (b) the average number of customers in the queue (excluding those in service) as a function of time. Then, we need statistical accumulators for the following: (i) number of customers who have started their service, to date, (ii) total waiting time for those customers, (iii) the integral, from time zero to the current simulation time, of the size of the waiting list as a function of time.

Let W_i denote the waiting time in the queue for customer i, $Q(t)$ the number of customers waiting in the queue at time t, $N_c(t)$ the number of customers who started their service during the time interval $[0, t]$, and let $N_c = N_c(T)$, where T is the time horizon. Here, T is deterministic and N_c is a random variable. The statistics in (a) and (b) above are precisely

$$\bar{W}_{N_c} = \frac{1}{N_c} \sum_{i=1}^{N_c} W_i$$

and

$$\bar{Q}_T = \frac{1}{T}\int_0^T Q(t)dt.$$

Those customers who arrive while the server is idle will have their W_i equal to zero, but are nevertheless counted in the average. In particular, if the system starts empty, one has $W_1 = 0$. Since $Q(t)$ is piecewise-constant, the integral of $Q(t)$ is easy to compute: It is a sum of rectangular areas, and corresponds to the total area under the "curve", from 0 to T, in Figure **??**.

1.2 Programming the single-server queue in C: Event view

The program `QueueEv` in Figure **??** simulates the single-server queue for a fixed time horizon. Our program assumes that the interarrival and service time distributions in the model are exponential with means 10 and 9, respectively, and that the time horizon is 1000.

The section at the beginning of the program declares the global variables.The three event types are `Arrival`, `Departure`, and `EndOfSimulation`.

The event list and simulation clock are automatically managed behind the scenes by SIMOD, initialized by `Sim_Init`, started by `Sim_Start`, and stopped by `Sim_Stop`. `Sim_Time` always returns the current simulation time, i.e., the current value of the simulation clock.

The main program (bottom) first initializes the simulation engine, then schedules an `EndOfSimulation` event in 1000 time units, schedules the first arrival, and starts the simulation. The time until the first arrival is an exponential random variable with mean 10, generated using the random number generator `GenArr`.

Module `Rand` offers a data type called `Rand_Stream`, which represents a random number generator. One can declare many variables of that type and each such variable can be viewed as a *virtual* generator. Two random number generators are declared in the top section: `GenArr` is used to generate the times between successive arrivals and `GenServ` to generate the service times of the customers. `Rand1_Expon` returns an exponential random variable with mean given by its second parameter, generated using the generator specified by the first parameter. Each customer has its service time generated upon arrival.

SSC provides a predefined type `List_List`, and suppose that objects like customers can be inserted and removed from those lists. For example, `List_Insert (Cust, List_Last, WaitList)` inserts the customer `Cust` at the tail of the list of waiting customers `WaitList`, and `List_Remove (Cust, List_First, WaitList)` removes the first customer in the queue to put it in the variable `Cust`. In `QueueEv`, there are two lists, `WaitList` and `ServList`, as in our previous outline.

We also have two variables of type `Stat_Block`, which are statistical accumulators, or *probes*. Those probes typically come in two flavors in simulation packages, as in SSC. The `Stat_Tally` type (like `CustWaits`) is appropriate when the statistical data of interest consist

```
#include <stdlib.h>
#include <stdio.h>
#include "Sim.h"
#include "Rand.h"
#include "Rand1.h"
#include "List.h"
#include "Stat.h"
#include "Event.h"

static Rand_Stream GenArr, GenServ;
static List_List WaitList, ServList;
static Stat_Block CustWaits, TotWait;
static Event_Type Arriv, Depart, EndOfSim;
struct Record { double ArrivTime, ServTime; };
static struct Record * Cust;

static void Arrival (void);
static void Departure (void);
static void EndOfSimulation (void);

static void Arrival (void)
  {
  Cust = (struct Record *) malloc (sizeof (struct Record));
  Cust->ArrivTime = Sim_Time ();
  Cust->ServTime = Rand1_Expon (GenServ, 9.0);
  if (List_Size (ServList) > 0)
    {
    List_Insert (Cust, List_Last, WaitList);
    Stat_Update (TotWait, List_Size (WaitList));
    }
  else
    {
    List_Insert (Cust, List_Last, ServList);
    Event_Schedule (Depart, Cust->ServTime, NULL);
    Stat_Update (CustWaits, 0.0);
    }
  Event_Schedule (Arriv, Rand1_Expon (GenArr,10.0), NULL);
  }


static void Departure(void)
  {
  Cust = List_Remove (List_Last, ServList);
  free (Cust);
  if (List_Size (WaitList) > 0)
    {
    Cust = List_Remove (List_First, WaitList);
    List_Insert (Cust, List_Last, ServList);
    Event_Schedule (Depart, Cust->ServTime, NULL);
    Stat_Update (CustWaits, Sim_Time () - Cust->ArrivTime);
    Stat_Update (TotWait, List_Size (WaitList));
    }
  }
```

Figure 1: Event-view simulation of a single-server queue

```
static void EndOfSimulation(void)
  {
  Stat_Report (CustWaits);
  Stat_Report (TotWait);
  Sim_Stop ();
  }

int main (void)
  {
  GenArr = Rand_CreateDefaultStream ("Generateur d'arrivees");
  GenServ = Rand_CreateDefaultStream ("Generateur de temps de service");
  WaitList = List_Create ("File d'attente");
  ServList = List_Create ("File de service");
  CustWaits = Stat_Create (Stat_Tally, "Temps d'attente des clients");
  TotWait = Stat_Create (Stat_Accumulate, "Nombre de clients dans la liste");
  Arriv = Event_Create (Arrival, "Arrivee");
  Depart = Event_Create (Departure, "Depart");
  EndOfSim = Event_Create (EndOfSimulation, "Fin de la simulation");
  Event_Schedule (EndOfSim, 100000.0, NULL);
  Event_Schedule (Arriv, Rand1_Expon (GenArr,10.0), NULL);
  Sim_Start ();
  return 0;
  }
```

Figure 1: Event-view simulation of a single-server queue (continuing)

of a sequence of observations $X_1, X_2, \ldots$ of which we might want to compute the sample mean, variance, and so on. Every update of the *tally* probe brings a new observation X_i. In the program of Figure **??**, for example, the block `CustWaits` is updated everytime a customer starts it service, by giving a new observation that corresponds to the waiting time W_i of that customer (the current time minus its arrival time). The total number of observations W_i at the end of the simulation will be $N_c = N_c(T) = N_c(1000)$. The `Stat_Accumulate` type of block (like `TotWait`), on the other hand, is used to integrate or compute the time-average for a continuous-time stochastic process with piecewise-constant trajectory. Whenever the value of the stochastic process changes in time (this may occur only at event times), the procedure `Stat_Update` should be called to give the new value, and the system then updates the current value of the integral by adding the product of the previous value of the process by the elapsed time since the last update. In our program, the block `TotWait` is updated every time the size of the waiting list changes. A certain (hidden) accumulator in that probe is equal, after each update, to the total waiting time of all the customers (that is, the integral of $Q(t)$) up to that instant.

Each customer is represented by a pointer to a record of information that contains its arrival time and service time. The variable `Cust` represents a generic customer. Each call to `malloc` (at customer's arrival) creates a new copy of that record and puts its address in the variable `Cust`, whereas `free (Cust)` (at customer's departure) deletes it.

When an arrival occurs, a new customer is created, its arrival time is set to the current

simulation time, and its service time is generated from the exponential distribution with mean 9. If someone is already in service, the customer is inserted in the waiting list (the queue) and the statistical block that keeps track of the size of that list is updated. Otherwise, the customer is inserted in the server's list, its departure is scheduled in S time units where S is the customer's service time, and its waiting time (zero) is given as a new observation to the statistical probe for the waiting times. Arrivals are scheduled by a domino effect: An arrival schedules the next one in A time units, where A is an exponential with mean 10.

At a departure, the customer in service is removed from the list and is destroyed. If someone is waiting, then the first in the waiting list is removed and inserted in the server's list, and its departure is scheduled. The waiting time of that customer (the current time minus its arrival time) is computed and given as an observation to the corresponding block. The block that keeps track of the size of the waiting list is also updated.

The event `EndOfSimulation` prints statistical reports for the two blocks and stops the simulation.

1.3 Example B: One day in a bank

This example is taken from [10]. A bank employs three tellers. Each work at exactly the same rate and can handle exactly the same customers. Most days, all three tellers report for work. However, 15% of the days only two tellers are present, and 5% of the days only one teller shows up for work. So, on any given day, t tellers report to work with probability q_t, where $q_3 = 0.80$, $q_2 = 0.15$, and $q_1 = 0.05$.

The bank opens at 10:00 and closes at 15:00 (i.e., 3 P.M.). The customers arrive randomly, according to a Poisson process, with piecewise constant rate $\lambda(t)$, $t \geq 0$; this means that the number of arrivals in a time interval $(t_1, t_2]$ is a Poisson random variable with mean $\int_{t_1}^{t_2} \lambda(t)dt$ and that the numbers of arrivals in disjoint time intervals are independent random variables. The arrival rate $\lambda(t)$ is assumed constant at 0.5 customer per minute from 9:45 until 11:00 and from 14:00 until 15:00, and one customer per minute from 11:00 until 14:00. Those customers who arrive before 9:45 join a FIFO queue outside the door and wait for the bank to open. At 15:00, the door is closed, but all the customers already in will be served. Service starts at 10:00.

Customers form a FIFO queue for the tellers, with balking. An arriving customer will balk (walk out) with probability p_k if there are k customers ahead of him in the queue (not counting the people receiving service), where p_k is given by:

$$p_k = \begin{cases} 0 & \text{if } k \leq 5; \\ (n-5)/5 & \text{if } 5 < k < 10; \\ 1 & \text{if } k \geq 10. \end{cases}$$

The customer service times are independent and distributed according to an Erlang distribution with parameters 2 and 2, i.e., as the sum of two independent exponential random variables with mean one. We want to estimate the expected number of customers served in an arbitrary day, and the expected average wait for the customers served on an arbitrary day. We might also be interested in the effect of changing the number of tellers, changing their speed, and so on.

Figure **??** give a program to simulate this bank model. The program have events at the fixed times 9:45, 10:00, etc. At 9:45, the counters are initialized and the arrival process is started. The time until the first arrival, or the time between one arrival to the next one, is (tentatively) an exponential with a mean of 2 minutes. However, as soon as an arrival turns out to be past 11:00, its time must be reajusted to take into account the increase of the arrival rate at 11:00. The event 11:00 takes care of this reajustment, and the event at 14:00 makes a similar reajustment when the arrival rate decreases. We give the specific name `Prochain` to the next planned arrival event to be able to reschedule that particular event (or process) to a different time. At the bank opening at 10:00, an event generates the number of tellers and starts the service for the corresponding customers. The event at 15:00 cancels the next arrival.

Upon arrival, a customer checks if a teller is free. If so, one teller becomes busy and the customer generates its service time and schedules his departure, otherwise the customer

```
#include <stdlib.h>
#include <stdio.h>
#include "Sim.h"
#include "Event.h"
#include "Stat.h"
#include "Rand.h"
#include "Rand1.h"


#define Minute 1.6666666666667E-2
/* Une minute.                  */

static double D, U;          /* Tampons.                      */
static long NCais;           /* Nb. de caissiers.             */
static long NOcc;            /* Nb. de caissiers occupes.     */
static long NAtt;            /* Nb. de clients en attente.    */
static long NServis;         /* Nb. servis aujourd'hui.       */
static long Rep;             /* Repetition courante.          */
static double DelaiMoyen;    /* Moy. inter-arrivees.          */
static Event_Notice Prochain; /* Prochaine arrivee.           */
static Event_Type E9h45, E10h, E11h, E14h, E15h, Arrivee, Depart;
static Stat_Block StatServis, Attente, AttenteMoy;
static Rand_Stream GenArr, GenServ, GenTeller, GenBalk;


static void Proc9h45 (void)
  {
  NCais = 0;
  NOcc = 0;
  NAtt = 0;
  NServis = 0;
  DelaiMoyen = 2 * Minute;
  Prochain = Event_ScheduleNotice (Arrivee,
   Rand1_Expon (GenArr, DelaiMoyen), NULL);
  }


static void Proc10h (void)
  {
  U = Rand_RandU01 (GenTeller);
  NCais = 3;
  if (U < 0.2)
    {
    NCais = 2;
    if (U < 0.05) NCais = 1;
    }
  while (NAtt > 0 && NOcc < NCais)
    {
    NOcc++;
    NAtt--;
    Event_Schedule (Depart, Rand1_Expon (GenServ, Minute)
    + Rand1_Expon (GenServ, Minute), NULL);
    }
  }
```

Figure 1: Event-view simulation of the bank model.

```
static void Proc11h (void)
  {
  D = Event_NoticeTime (Prochain) - Sim_Time ();
  Event_CancelNotice (&Prochain);
  Prochain = Event_ScheduleNotice (Arrivee, D/2.0, NULL);
  DelaiMoyen = Minute;
  }


static void Proc14h (void)
  {
  D = Event_NoticeTime (Prochain) - Sim_Time ();
  Event_CancelNotice (&Prochain);
  Prochain = Event_ScheduleNotice (Arrivee, 2.0*D, NULL);
  DelaiMoyen = 2*Minute;
  }


static void Proc15h (void)
  {
  Event_CancelNotice (&Prochain);
  }


static int Balk(void)
  {
  return ( (NAtt > 9) || ( (NAtt > 5)
           && (5 * Rand_RandU01 (GenBalk) < NAtt-5) ) );
  }


static void ProcArrivee (void)
  {
  Prochain = Event_ScheduleNotice (Arrivee,
   Rand1_Expon (GenArr, DelaiMoyen), NULL);
  if (NOcc < NCais)
    {
    NOcc++;
    Event_Schedule (Depart,Rand1_Expon (GenServ, Minute)
                         + Rand1_Expon (GenServ, Minute), NULL);
    }
  else if (!Balk())
    {
    NAtt++;
    Stat_Update (Attente,NAtt);
    }
  }
```

Figure 1: Event-view simulation of the bank model (continuation).

```
static void ProcDepart (void)
  {
  NServis++;
  if (NAtt > 0)
    {
    NAtt--;
    Event_Schedule (Depart, Rand1_Expon (GenServ, Minute)
                          + Rand1_Expon (GenServ, Minute), NULL);
    Stat_Update (Attente, NAtt);
    }
  else
    NOcc--;
  }


static void UnJour (void)
   {
   Sim_Init ();
   Stat_Init (Attente);
   Event_Schedule (E9h45, 9.75, NULL);
   Event_Schedule (E10h, 10.0, NULL);
   Event_Schedule (E11h, 11.0, NULL);
   Event_Schedule (E14h, 14.0, NULL);
   Event_Schedule (E15h, 15.0, NULL);
   Sim_Start ();
   Stat_Update (StatServis, NServis);
   Stat_Update (AttenteMoy, Stat_Sum (Attente));
   }


int main (void)
  {
  GenArr = Rand_CreateDefaultStream ("Generateur d'arrivees");
  GenServ = Rand_CreateDefaultStream ("Generateur de temps de service");
  GenTeller = Rand_CreateDefaultStream ("Generateur du nombre de caissiers");
  GenBalk = Rand_CreateDefaultStream ("Generateur pour le balking");
  E9h45 = Event_Create (Proc9h45, "Demarrage");
  E10h = Event_Create (Proc10h, "Ouverture");
  E11h = Event_Create (Proc11h, "11 heures");
  E14h = Event_Create (Proc14h, "14 heures");
  E15h = Event_Create (Proc15h, "Fermeture");
  Arrivee = Event_Create (ProcArrivee, "Arrivee d'un client");
  Depart = Event_Create (ProcDepart, "Depart d'un client");
  StatServis = Stat_Create (Stat_Tally, "Nb. servis par jour");
  Attente = Stat_Create (Stat_Accumulate, "Attente cumulee aujourd'hui (heures)");
  AttenteMoy = Stat_Create (Stat_Tally, "Attente moy. par jour (heures)");
  for (Rep = 1; Rep <= 100; Rep++)
    {
    UnJour();
    }
  Stat_Report (StatServis);
  Stat_Report (AttenteMoy);
  return 0;
  }
```

Figure 1: Event-view simulation of the bank model (continuation).

either balks or joins the queue. The balking decision is computed by the procedure `Balk`, using the random number generator `GenBalk`. The arrival event also generates the next scheduled arrival. Upon departure, the customer frees the teller, and the first customer in the queue, if any, can start its service.

The procedure `UnJour` initializes the clock, event list, and statistical probe for the waiting times, schedules the deterministic events, and run the simulation. After 15:00, no more arrival occurs and the event list becomes empty when the last customer departs. At that point, the program returns to right after the `Sim_Start` statement and updates the statistical counters for the number of customers served during the day and their total waiting time. The main program calls `UnJour` 100 times, to simulate 100 days of operation of the bank, and prints statistical reports. If X_i is the number of customers served on day i and Q_i the total waiting time on day i, the program estimates $E[X_i]$ and $E[Q_i]$ by their sample averages $\bar{X}_n$ and $\bar{Q}_n$ with $n = 100$.

Optbm

This module implements simulation tools for pricing financial options on a single asset, where the evolution of the price of the asset is modeled by a geometric brownian motion (GBM). Various types of options are considered.

We assume that the price $S(t)$ of the underlying asset evolves as a GBM motion with *drift* parameter α and variance (or *volatility*) parameter σ^2.

An *European call option* for that asset gives its owner the right to buy the asset at price K (the *strike price*) at time T (the *expiration date*). The net payoff of the call option at time T is $\max[0, S(T) - K]$, assuming that the asset provides no dividend.

In the idealized world of Black and Scholes [8], there is a unique probability law, called the *risk neutral* measure, under which $\{S(t),\, t \geq 0\}$ is a GBM with drift parameter r and variance parameter σ^2, and the value of the option at time 0, if $S(0) = s$, is

$$v(s,T) = e^{-rT}\, E[\max(0, S(T) - K) \mid S(0) = s] \tag{1}$$

where E denotes the expectation under the risk-neutral probability measure where the drift parameter is r. See, e.g., [19] for the details.

To compute $v(s,T)$, using the fact that $[\ln(S(t)/S(0)) - rt]/\sigma\sqrt{t}$ is a standard normal, it can be derived (see, e,g, [46] or [19]) that

$$\begin{aligned} v(s,T) &= e^{-rT} \int_{z_0}^{\infty} \left[s e^{(r-\sigma^2/2)T + z\sigma\sqrt{T}} - K \right] \phi(z) dz \\ &= s\Phi(-z_0 + \sigma\sqrt{T}) - K e^{-rT} \Phi(-z_0), \end{aligned} \tag{2}$$

where

$$z_0 = \frac{\ln(K/s) - (r - \sigma^2/2)T}{\sigma\sqrt{T}}$$

and ϕ and Φ are the density and the distribution function of the standard normal. Equation (**??**) is the celebrated *Black-Scholes formula* for option pricing.

An European call option is one of the simplest kind of option that one can buy on financial markets. An European *put* option is similar to a call, except that it gives the right to *sell* (instead of buy) the asset at price K. It can be priced by a formula similar to (**??**). For a whole bunch of more complicated options however, no simple pricing formula is available. Monte Carlo simulation offers a viable way of pricing these options.

For example, an *asian call option* based on the arithmetic average has a payoff at time T given by

$$\max\left(0,\, \frac{1}{t}\sum_{j=1}^{t} S(\tau_j) - K\right), \tag{3}$$

which is the difference between the average value of the asset at pre-determined observation times $\tau_1, \ldots, \tau_t$ and the strike price K, if this value is positive. Otherwise the payoff is 0.

The *value* of the option at time 0 is the mathematical expectation of that payoff multiplied by the discount factor e^{-rT}, conditional on $S(0)$, under the risk-neutral measure.

No analytic formula is available for this expectation but it can be estimated by simulation as follows. We can write

$$S(\tau) = S(0)\exp[(r - \sigma^2/2)\tau + \sigma B(\tau)], \tag{4}$$

where $\{B(\tau), \tau \geq 0\}$ is a standard Brownian with $B(0) = 0$. The values of $B(\cdot)$ at the observation dates can be simulated by generating t i.i.d. $N(0,1)$ random variables $Z_1, \ldots, Z_t$ and putting, recursively, $B(\tau_j) = B(\tau_{j-1}) + (\tau_j - \tau_{j-1})Z_j$, where $\tau_0 = 0$. Alternatively, they can be generated via the Brownian bridge approach. The payoff discounted to time 0 is

$$e^{-rT}\max\left(0,\ \frac{1}{t}\sum_{j=1}^{t} S(\tau_j) - K\right), \tag{5}$$

where the $S(\tau_j)$'s are computed via (**??**). This can be replicated, say, n times, independently, and the option value is estimated by the average discounted payoff. This is a straighforward or *crude* Monte Carlo approach. An *asian put option*, whose payoff at time T is

$$\max\left(0,\ K - \frac{1}{t}\sum_{j=1}^{t} S(\tau_j)\right), \tag{6}$$

can be evaluated in the same way.

More efficient estimators for this model, and for other kinds of options, are discussed in [33].

```
#include "Rand.h"
#include "Stat.h"


struct Optbm_InfoOpt
   {
   double T, T1, t, delta, K, S0, r, sigma;
   };
typedef struct Optbm_InfoOpt * Optbm_Opt;
```

Parameters of an Asian option. The observations dates are $\tau_1, \ldots, \tau_t$ where $\tau_i = T_1 + i\delta$ and $\delta = (T - T_1)/t$ (so $\tau_t = T$). The strike price is K, the initial value is S_0, the risk-free rate is r and the volatility is σ.

```
struct Optbm_InfoExper
   {
   long         nrepmc, nrepqmc, nptqmc, t;
   int          qmc, av, cv;
   Rand_Stream  RandOpt, Randqmc;
   double       PayoffA, PayoffG;
   Stat_Block   StatVal, StatValG;
   };
typedef struct Optbm_InfoExper * Optbm_Exper;
```

Parameters of a simulation experiment for pricing an option. The variables `nrepmc`, `nrepqmc`, and `nptqmc` indicate the number of independent simulation runs for the MC

method, the number of independent replications (random shifts or xors) for the QMC method, and the number of t-dimensional points for each QMC point set, respectively. To obtain the same number of simulation runs for both MC and QMC, one should have `nrepmc = nrepqmc * nptqmc`.

The boolean `qmc` should be 1 if we use the QMC method, 0 otherwise. `av` indicates if we use antithetic variates. When `av = 1`, the total number of runs is doubled, because the antithetic runs are obtained almost for free, in addition to the variance reduction obtained by inducing a negative correlation. `cv` indicates if we use the payoff of the option based on the geometric average as a control variable.

`RandOpt` and `Randqmc` are the random number streams used to simulate the option and to randomize the point set when QMC is used, respectively.

The variables `PayoffA` and `PayoffG` are used to store the payoff of the option based on the arithmetic and geometric averages, respectively, on a given simulation run. `StatVal` collects the observations of the payoff from independent simulation runs, whereas `StatValG` collects the observations for the geometric average (if `cv = 1`).

```
Optbm_Opt Optbm_CreateOpt (long t, double T, double T1, double K,
                           double S0, double r, double sigma);
```

Returns a new structure for an Asian option with the specified parameters. Also computes the parameter `delta`.

```
void Optbm_DeleteOpt (Optbm_Opt Opt);
```

Deletes the Asian option `Opt` (recovers the memory).

```
Optbm_Exper Optbm_CreateExper (
   long nrepmc, long nrepqmc, long nptqmc, long t,
   int qmc, int av, int cv,
   Rand_Stream RandOpt, Rand_Stream Randqmc);
```

Returns a new structure for a simulation experiment with the specified parameters. The random number streams `RandOpt` and `Randqmc` must have been created before.

```
void Optbm_DeleteExper (Optbm_Exper Exp);
```

Deletes the structure `Exp` (recovers the memory).

```
double Optbm_ValueGeo (Optbm_Opt Opt);
```

Returns the value of an Asian option with parameters in `Opt`, based on the geometric average.

```
void Optbm_SimPayoff (Optbm_Opt Opt, Optbm_Exper E);
```

Simulates the process for an Asian option with parameters in `Opt`, with the experiment parameters given in `E`, for a single run. Returns the payoff based on the arithmetic average in `PayoffA`. If `cv = 1`, returns also the payoff based on the geometric average in `PayoffG`. If `av = 1`, these payoffs are the average over the regular and antithetic values. This procedure uses t numbers from the stream `E->RandOpt`, i.e., a single point of the QMC point set in the case of QMC.

`void Optbm_SimExper (Optbm_Opt Opt, Optbm_Exper E);`

Performs a simulation experiment for an option with parameters in `Opt`, with the experiment parameters given in `E`. Makes `nrepmc` independent runs in the case of MC, and uses `nrepqmc` independent random shifts of a point set of size `nptqmc` in the case of QMC. In both cases, the independent observations are placed in the statistical block `E->StatVal`.

`void Optbm_ReadParam (FILE *infile, Optbm_Opt *pO, Optbm_Exper *pE);`

Reads the parameters for the option `*pO` and for the simulation experiment `*pE` in the data file `infile`.

`void Optbm_WriteParam (Optbm_Opt Opt, Optbm_Exper Exp);`

Writes the parameters for the option `Opt` and for the simulation experiment `Exp` on standard output.

`void Optbm_WriteResults (Optbm_Opt Opt, Optbm_Exper E);`

Prints the results of a simulation experiment `E` made with the option `Opt`.

APPENDIX A

Functional definition of SSC

Rand

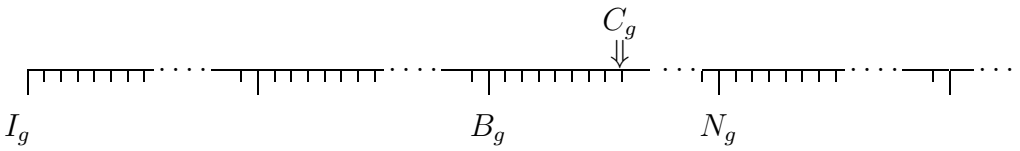
This module defines the basic structures to handle multiple streams of uniform (pseudo)-random numbers and convenient tools to move around within and across these streams.

The actual random number generators (RNGs) are implemented separately in other modules, and a variety of such RNGs are available, e.g., in modules `Mrg32k3, Combtaus`, ... The random number streams can also be instantiated as points from HUPS (see module `Qrand`).

Each stream of random numbers is an object of the class `Rand_Stream`, and can be viewed as a virtual random number generator. A stream must be declared and created before being used, and can be deleted when no longer needed. The creation and deletion is done by calling procedures from the modules that implement specific classes of RNGs and HUPS. When generating a random variate from a given distribution, e.g., by the procedures in modules `Rand1, Rand2, Rand3`, one must indicate the `Rand_Stream` that provides the underlying uniform random numbers.

For each type of RNG (not HUPS) provided in SSC, the full period of the generator is cut into adjacent *streams* (or segments) of length Z, and each of these streams is partitioned into V *blocks* (or substreams) of length W, where $Z = VW$. The values of V and W depend on the specific RNG, but are usually larger than 2^{50}. Thus, the distance Z between the starting points of the successive streams provided by an RNG usually exceeds 2^{100}. The initial seed of the RNG is the starting point of the first stream. It has a default value in the module for each type of RNG, but its value can be changed by the user via a simple procedure call. Each time a new stream (i.e, an object of the class `Rand_Stream`) is created, the starting point (initial seed) of this stream is computed automatically, Z steps ahead of the starting point of the previously created stream, and its current *state* is set equal to this starting point.

For each stream, one can generate one value and go ahead one step, or go ahead to the beginning of the next block within this stream, or go back to the beginning of the current block, or to the beginning of the stream, or jump ahead or back by an arbitrary number of steps. Denote by C_g the current state of a stream g, I_g its initial state, B_g the state at the beginning of the current block, and N_g the state at the beginning of the next block. The following diagram shows an example of a stream whose state is at the 6th value of the third block, i.e., $2W + 5$ steps ahead of its initial state I_g and 5 steps ahead of its state B_g.



The procedures for manipulating the streams and generating random numbers are implemented differently for each type of RNG. Pointers to some of them (those whose parameter types do not depend on the RNG type) are available in the structure `Rand_InfoStream`.

The other (e.g., for setting the seeds) are available only in the modules that implement the specific RNG types. We now briefly describe these procedures.

The procedure `Rand_ResetStream` resets a given stream either to its initial state ($C_g \leftarrow I_g$ and $B_g \leftarrow I_g$), or to the beginning of its current block ($C_g \leftarrow B_g$), or to the beginning of its next block ($C_g \leftarrow N_g$ and $B_g \leftarrow N_g$). The procedures `Rand_RandU01` and `Rand_RandInt` generate the uniform (pseudo)random numbers. Each stream can produce (if desired) *antithetic* random numbers with respect to the uniforms normally produced, i.e., $1 - U$ instead of U, by calling `Rand_SetAntithetic` to turn a switch on and off.

The procedure `GetState` returns the state of a stream. One can change the state of a given stream, without modifying that of the other streams, by calling `SetSeed`. However, after calling `SetSeed` for a given stream, the initial states of the different streams are no longer spaced Z values apart. Therefore, this procedure should be called very sparingly, if at all. The procedure `ResetStream` suffices for almost all applications. In certain RNG modules, one may use the procedure `DoubleGenerator` to get double precision numbers upon calling `RandU01`.

Here are two examples of situations where the tools offered in this package are useful.

(a) You want to compare two or more similar systems, via simulation with common random numbers, with n simulation runs for each system. To guarantee that the same random numbers are used across the systems with good synchonization, assign different streams to different places where random numbers are needeed in the model (e.g., to compare queueing systems, use one stream for the interarrival times, one stream for the service times at each queue, one stream for routing decisions, etc.). To make sure that each stream starts from the same state across the different systems, assign run i to the ith block, for each stream. The experiment then proceeds as follows. For the first system, simulate run 1 by starting all the streams from their initial seed, I_g. Before each new run, advance all the streams to the initial state of their next block, N_g. After the nth run, reset all the streams to their initial state, I_g. Repeat for each system that you want to compare.

(b) You want to run simulations on several processors, in parallel, and you want each processor to have its own (virtual) random number generator. In this case, you can simply use one stream for each processor. You need a central authority, or *monitor*, to manage the allocation of the streams, but once a stream is allocated, the processors can go their own way, independently of each other. In this setup, the processors use in fact the same generator, but with different *seeds*. This makes the implementation easier than if a different generator is used on each processor.

```
typedef enum {Rand_StartStream, Rand_StartBlock,
      Rand_NextBlock} Rand_SeedType;

typedef struct Rand_InfoType * Rand_StreamType;
struct Rand_InfoType
   {
   char * name;
   void (*SetAntithetic) (void *, int);
   void (*ResetStream) (void *, Rand_SeedType);
   void (*WriteState) (void *);
   double (*RandU01) (void *);
   long (*RandInt) (void *, long, long);
   };

typedef struct Rand_InfoStream * Rand_Stream;
struct Rand_InfoStream
   {
   char * name;
   Rand_StreamType Type;
   void * State;
   };

Rand_Stream Rand_CreateDefaultStream (char * name);
```

Creates and returns a new default stream. This procedure reserves space to keep the information relative to the `Rand_Stream`, sets its antithetic switch to 0. The default random-generator is `Mrg32k3`. (See module `Mrg32k3` for other details.)

```
void Rand_ResetStream (Rand_Stream g, Rand_SeedType where);
```

Reinitialize the current state of stream `g`, according to the value of `where`:
If (`where == Rand_StartStream`), C_g and B_g are set to I_g;
if (`where == Rand_StartBlock`), C_g is set to B_g;
if (`where == Rand_NextBlock`), N_g is computed, and C_g and B_g are set to N_g.

```
void Rand_SetAntithetic (Rand_Stream g, int a);
```

When this procedure is called with `a` $\neq$ `0`, the stream `g` starts generating antithetic variates, i.e., $1 - U$ instead of U, until the procedure is called again with `a` $=$ `0`.

```
void Rand_WriteState (Rand_Stream g);
```

Prints (to standard output) the current state of `g`.

```
double Rand_RandU01 (Rand_Stream g);
```

Returns a (pseudo)random number from the uniform distribution over the interval $(0, 1)$, using stream `g`. For RNG, the state of the stream `g` is advanced before taking the (pseudo)random number. For HUPS (`Qrand`), the state is advanced after. [3]

[3] From Pierre: Should advance after instead (e.g., for Qrand)?

`long Rand_RandInt (Rand_Stream g, long i, long j);`

Returns a (pseudo)random number from the discrete uniform distribution over the integers $\{i, i+1, \ldots, j\}$, using stream `g`.For RNG, the state of the stream `g` is advanced before taking the (pseudo)random number. For HUPS (`Qrand`), the state is advanced after.

Mrg32k3

This is an implementation of a random number generator as described in the module `Rand`. The backbone (or main) generator is the combined multiple recursive generator (CMRG) `MRG32k3a` proposed by L'Ecuyer [32]. The generator is implemented in 64-bit floating-point arithmetic. This is faster than an integer implementation (as in [31]) on most current computers, for which the floating-point operations are implemented in hardware on the main chip.

The backbone generator has period length $\rho \approx 2^{191}$. The values of V, W, and Z are 2^{51}, 2^{76}, and 2^{127}, respectively. The seed of the RNG, and the state of a stream at any given step, are vectors of 6 32-bit integers. The default initial seed of the RNG is $(12345, 12345, 12345, 12345, 12345, 12345)$.

```
typedef struct Mrg32k3_InfoState * Mrg32k3_StreamState;
struct Mrg32k3_InfoState
   {
   double Cg[6], Bg[6], Ig[6];
   int Anti;
   int DoubleGen;
   };
```

The state of a stream from the present module. The arrays `Cg`, `Bg`, and `Ig` contain the current state, the starting point of the current block in the state, and the starting point of the stream. This stream generates antithetic variates if and only if `Anti = 1`. The precision of the output numbers is "doubled" (see `Mrg32k3_DoubleGenerator`) if and only if `DoubleGen = 1`.

```
Rand_Stream Mrg32k3_CreateStream (char * name);
```

Creates and returns a new stream. This procedure reserves space to keep the information relative to the `Rand_Stream`, initializes its seed I_g, sets B_g and C_g equal to I_g, sets its antithetic switch to 0. The seed I_g is equal to the initial seed of the package given by `Mrg32k3_SetPackageSeed` if this is the first stream created, otherwise it is Z steps ahead of that of the most recently created stream.

```
void Mrg32k3_DeleteStream (Rand_Stream * g);
```

Deletes the stream `g`, if it has been created previously by `Mrg32k3_CreateStream`, and recover its memory. Otherwise does nothing.

```
void Mrg32k3_ResetStream (Mrg32k3_StreamState S, Rand_SeedType where);
```

Reinitialize the current state of stream with state `S`, according to the value of `where` (see `Rand_ResetStream`).

```
void Mrg32k3_DoubleGenerator (Mrg32k3_StreamState S, int d);
```

After calling this procedure with `d`=1, each call to `Mrg32k3_RandU01` for stream with state `S` (direct or indirect) will return a number with 53 bits of precision instead of 32 bits, and will

advance the state of the stream by 2 steps instead of 1. More specifically, in the non-antithetic case, the instruction "`x = Mrg32k3_RandU01(g)`" when generator is "doubled" is equivalent to "`x = (Mrg32k3_RandU01(g) + Mrg32k3_RandU01(g) * fact) % 1.0`" where the constant `fact` is equal to 2^{-24}. This also applies when calling `Mrg32k3_RandU01` indirectly, e.g., by calling `Mrg32k3_RandInt`.

By default, or if this procedure is called again with `d`=0, each call to `Mrg32k3_RandU01` for stream `g` returns a number with 32 bits of precision and advances the state by 1 step.

```
void Mrg32k3_SetAntithetic (Mrg32k3_StreamState S, int a);
```

When this procedure is called with `a` $\neq$ `0`, the stream with state `S` starts generating antithetic variates, i.e., $1 - U$ instead of U, until the procedure is called again with `a` = `0`.

```
void Mrg32k3_SetPackageSeed (unsigned long seed[6]);
```

Sets the initial seed of the package `Mrg32k3` to the six integers in the vector `seed`. This will be the seed (initial state) of the first stream. If this procedure is not called, the default initial seed is $(12345, 12345, 12345, 12345, 12345, 12345)$. If it is called, the first 3 values of the seed must all be less than $m_1 = 4294967087$, and not all 0; and the last 3 values must all be less than $m_2 = 4294944443$, and not all 0.

```
void Mrg32k3_SetSeed (Mrg32k3_StreamState S, unsigned long seed[6]);
```

Sets the initial seed I_g of stream with state `S` to the vector `seed`. This vector must satisfy the same conditions as in `Mrg32k3_SetPackageSeed`. The stream is then reset to this initial seed. The states and seeds of the other streams are not modified. As a result, after calling this procedure, the initial seeds of the streams are no longer spaced Z values apart. We discourage the use of this procedure. Proper use of `Mrg32k3_ResetStream` is preferable.

```
void Mrg32k3_AdvanceState (Mrg32k3_StreamState S, long e, long c);
```

Advances the state of stream with state `S` by k values, without modifying the states of other streams (as in `Mrg32k3_SetSeed`), nor the values of B_g and I_g associated with this stream If $e > 0$, then $k = 2^e + c$; if $e < 0$, then $k = -2^e + c$; and if $e = 0$, then $k = c$. Note: c is allowed to take negative values.

```
void Mrg32k3_GetState (Mrg32k3_StreamState S, unsigned long seed[6]);
```

Returns the current state C_g of stream with state `S`. This is convenient if we want to save the state for subsequent use.

```
void Mrg32k3_WriteState (Rand_Stream g);
```

Prints (to standard output) the current state C_g of stream `g` with state `S`.

```
double Mrg32k3_RandU01 (Mrg32k3_StreamState S);
```

Returns a (pseudo)random number from the uniform distribution over the interval $(0, 1)$, using stream with state `S`, then advances the state by one step. The returned number has 32 bits of precision, unless the stream had been "doubled" by calling `Mrg32k3_DoubleGenerator`.

`long Mrg32k3_RandInt (Mrg32k3_StreamState S, long i, long j);`

Returns a (pseudo)random number from the discrete uniform distribution over the integers $\{i, i+1, \ldots, j\}$, using stream with state `S`. Makes one call to `Mrg32k3_RandU01`.

Qrand

This module provides basic structure and tools to define random number streams that are in fact implemented as randomized low-discrepancy point sets. The type of point set considered by this module is a point set that comprises all the points formed by successive values of a periodic recurrence, from all its possible initial states. Specific implementations of such point sets are available from the modules `Qlcg`, `Qtaus`, etc., which provide various types of lattice rules. The cycle set structures available here (see `Qrand_CycleSet`) must be created and filled up from these modules. A stream can then be created by `Qrand_CreateStream`, by associating it with a given cycle set. Note that several different streams can be created based on the same cycle set. These different streams will usually be randomized independently (by calling `Qrand_Randomize`), so their output sequences will differ.

```
typedef enum {Qrand_None, Qrand_RandomShift,
      Qrand_RandomXor} Qrand_RandomType;

typedef struct Qrand_InfoCycle * Qrand_Cycle;
struct Qrand_InfoCycle
   {
   long Length;
   tables_TabLR ValLR;
   tables_TabUL ValUL;
   Qrand_Cycle Next;
   };

typedef struct Qrand_InfoCycleSet * Qrand_CycleSet;
struct Qrand_InfoCycleSet
   {
   long NbCycles;
   long NbPoints;
   Qrand_Cycle FirstCycle;
   };
```

A `Qrand_CycleSet` is a pointer to a structure that contains the successive values produced by a periodic recurrence, over all of its cycles. These values are organized into all the cycles that can be obtained, from all possible initial seeds. `NbCycles` is the number of distinct cycles, `NbPoints` is the total number of points in all the cycles, `FirstCycle` points to the first cycle.

`Qrand_Cycle` is a pointer to a cycle. For each cycle, `Length` is the cycle length and `ValLR[0..Length-1]` or `ValUL[0..Length-1]` contains the `Length` successive values in the cycle. The rule of the cycle set decides witch array to use between `ValLR` (e.g. `Qlcg`) and `ValUL` (e.g. `Qtaus`). These values are real numbers (`ValLR`), usually in the interval $[0, 1)$, or unsigned integer (`ValUL`), usually a 32-bit representation . Each cycle points to the next one (`Next`), and the last cycle points to the first one.

```
typedef struct Qrand_InfoState * Qrand_StreamState;
struct Qrand_InfoState
   {
   Qrand_CycleSet C;
   Qrand_Cycle PtrCurCycle;
   long CurCycle;
   long CurPos;
   long CurUnif;
   Qrand_RandomType R;
   long SizeRandom;
   Rand_Stream GRandom;
   tables_TabLR Rshift;
   tables_TabUL Rxor;
   int Anti;
   };
```

`Qrand_StreamState` is a pointer to a structure that contains the state of a `Rand_Stream` defined in terms of a cycle set `C`. The variables `CurCycle`, `CurPos`, and `CurUnif` indicate the current cycle, the current position in the cycle (i.e., the start of the current block or point), and the current value in the block (i.e., the current coordinate, which is the index of the next output value in the current cycle), respectively. All of theses variables are initialized to 0. (See also `Qrand_ResetStream`.) `PtrCurCycle` points to the current cycle. `R` indicates which type of randomization of the point set is used for this stream. The table `Rshift` or `Rxor`, created and filled up by calling `Qrand_Randomize`, holds the uniforms used for the randomization. `SizeRandom` is the number of uniforms in the table (i.e., the dimension that is filled up). `GRandom` is the RNG used to fill `Rshift` or `Rxor`. The field `Anti` gives the current value of the antithetic switch.

```
void Qrand_DeleteCycleSet (Qrand_CycleSet * C);
```

Deletes the structure `C` and recovers the memory used to store the list of cycles and the values.

```
Rand_Stream Qrand_CreateStream (char * name, Qrand_CycleSet C,
                                Qrand_RandomType R);
```

Creates and returns a new stream based on the cycle set `C`, which must have been created and filled up before. This procedure creates and initializes a `Qrand_StreamState` structure which holds the state of this stream. The stream is set to `Rand_InitStream` and its antithetic switch is set to OFF. `R` specifies which type of randomization is applied to the points defined by `C`. The vector `Rshift` or `Rxor` used for the randomization is not created at this stage; it is created afterwards, at the first call to `Qrand_Randomize`.

```
void Qrand_DeleteStream (Rand_Stream * g);
```

Deletes the stream `g` that has been created previously by `Qrand_CreateStream`, and recover its memory. It does not delete the corresponding cycle set `C`.

```
void Qrand_ResetStream (Qrand_StreamState S, Rand_SeedType where);
```

Reinitializes the current state of the stream with state `S`, according to the value of `where` (see `Rand_ResetStream`). This is achieved by setting `CurCycle`, `CurPos`, and `CurUnif` appropriately. In all cases, `CurUnif` is reset to 0. If `where = Rand_StartStream`, `CurCycle` and `CurPos` are set to 0 and 0, respectively. If `where = Rand_NextBlock`, `CurPos` is increased by 1 if it is not

at the end of the current cycle (i.e., `CurPos < Length`), otherwise `CurCycle` is increased by 1 and `CurPos` is set to 0.

`void Qrand_WriteState (Rand_Stream g);`

Prints (to standard output) the current state of stream `g`.

`void Qrand_SetAntithetic (Qrand_StreamState S, int a);`

When this procedure is called with `a` $\neq$ `0`, the stream with state `S` starts generating antithetic variates, i.e., $1 - U$ instead of U, until the procedure is called again with `a` $=$ `0`.

`void Qrand_Randomize (Qrand_StreamState S, Rand_Stream g0, long t);`

Generates a random vector to be used for the randomization (random shift or xor) of the point set associated with stream `g`. This vector is generated using the existing stream `g0`. The parameter `t` is the dimension of the random vector. If the actual number of successive calls to the generator within a given block eventually exceeds `t`, the size of this random vector will be extended automatically with the stream `g0` (work if g0 is not deleted).

`double Qrand_RandU01 (Qrand_StreamState S);`

Returns a number in the interval $[0, 1)$ from the stream that has state `S`, then advances its current state by one step.

`long Qrand_RandInt (Qrand_StreamState S, long i, long j);`

Returns an integer "quasi-uniformly" distributed over the integers $\{i, i+1, \ldots, j\}$, using stream that has state `S`, then advances its current state by one step.

`tables_TabLR Qrand_RandU01t (Qrand_StreamState S, long t);`

Returns a random point generated over the unit hypercube $[0, 1)^t$ from the stream that has state `S`. The current state is advanced by 1, as when calling `Qrand_RandU01`. (see module tables in library MYLIB-C)

Qlcg

This module provides Korobov-type lattice rules, based on a LCG with prime or power-of-2 modulus, for multidimensional integration. The underlying theory is described in [34, 41].

— Références. Randomisations. Critères.

How to use this module. One should first call one of the procedures `Qlcg_Select*` or `Qlcg_Choose*` to determine a rule. With `Qlcg_Choose*`, the user must choose explicitly the parameters of the rule, whereas with `Qlcg_Select*`, the package selects the parameters automatically from precomputed tables, according to the specified criterion.

Note : The FirstCycle (cycle no:0) is the cycle with only the value zero (length = 1). The seed of the others cycles is an unvisited state.

```
typedef enum {Qlcg_M8, Qlcg_M32, Qlcg_M32x24x16x12} Qlcg_Crit;
```

Selection criteria for the lattice rules...

```
Qrand_CycleSet Qlcg_ChoosePrim (long n, long a);
```

Creates and returns a `Qrand_CycleSet` structure and fills it by a Korobov lattice rule based on a LCG with prime modulus n and primitive multiplier a.

```
Qrand_CycleSet Qlcg_SelectPrim (Qlcg_Crit Crit, long e, long * n, long * a);
```

The package selects and returns the values of n and a, and returns `Qlcg_ChoosePrim (n, a)`. The value of n is the largest prime number less than 2^e, and the multiplier a is chosen based on the criterion `Crit`.

```
Qrand_CycleSet Qlcg_ChoosePow2 (long e, long a);
```

Creates and returns a `Qrand_CycleSet` structure and fills it by a Korobov lattice rule based on a LCG with prime modulus $n = 2^e$ and multiplier a.

```
Qrand_CycleSet Qlcg_SelectPow2 (Qlcg_Crit Crit, long e, long * a);
```

The package selects and returns the value of a based on `Crit`, and returns `Qlcg_ChoosePow2 (e, a)`.

```
void Qlcg_WriteCycleSet (Qrand_CycleSet C);
```

Writes the parameters associated with the cycle set `C`.

Qtaus

This module provides polynomial lattice rules based on linear recurrences modulo 2, implemented in the form of Tausworthe random number generators. The underlying theory is described in [34, 36].

How to use this module. One should first call the procedures `Qtaus_Select*` to determine a rule. With `Qtaus_Select*`, the package selects the parameters automatically from precomputed tables, according to the specified criterion.

Note : The FirstCycle (cycle no:0) is the cycle with only the value zero (length = 1).

```
typedef enum {Qtaus_D8, Qtaus_D32} Qtaus_Crit;
```

Selection criteria for point sets defined via combined Tausworthe recurrences.

```
Qrand_CycleSet Qtaus_SelectComb2 (Qtaus_Crit Crit, long e);
```

The package selects a combined Tausworthe recurrence with 2 components, based on the criterion `Crit`, to define a point set with $n = 2^e$ points. The FirstCycle (or cycle no: 0) corresponds to not combine any components. The 3 others cycles correspond to the different combinations (all having maximal period) from the 2 components. It then creates, fills up and returns a `Qrand_CycleSet` with the corresponding cycles.

```
Qrand_CycleSet Qtaus_SelectComb3 (Qtaus_Crit Crit, long e);
```

The package selects a combined Tausworthe recurrence with 3 components, based on the criterion `Crit`, to define a point set with $n = 2^e$ points. The FirstCycle (cycle no: 0) corresponds to not combine any components. The 7 others cycles correspond to the different combinations (all having maximal period) from the 3 components. It then creates, fills up and returns a `Qrand_CycleSet` with the corresponding cycles.

```
void Qtaus_WriteCycleSet (Qrand_CycleSet C);
```

Writes the parameters associated with the cycle set `C`.

Fdist

This module provides procedures that approximate certain distributions functions and their inverses.

`double Fdist_Normal1 (double x);`

Returns an approximation of $\Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses the approximation given in [27, page 90].

`double Fdist_Normal2 (double x);`

Returns an approximation of $\Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses the Tchebychev approximation proposed by [40], which gives 16 decimals of precision over the entire real line.

`double Fdist_Normal3 (double x);`

Returns an approximation of $\Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses the `erf` function from the standard C library (`man erf`).

`double Fdist_FBarNormal (double x);`

Returns an approximation of $1 - \Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses a Chebyshev series giving 16 decimal digits of precision (see [40]).

`double Fdist_InvNormal1 (double u);`

Returns an approximation of $\Phi^{-1}(u)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses a rational approximation giving at least 7 decimal digits of precision when $10^{-20} < u < 1 - 10^{-20}$ (see [10, 27]).

`double Fdist_InvNormal2 (double u);`

Returns an approximation of $\Phi^{-1}(u)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses minimax rational approximations giving at least 16 decimal digits of precision (see [**?**]).

`double Fdist_InvStudent (long n, double u);`

Returns an approximation of $F^{-1}(u)$, where F is the Student distribution function with n degrees of freedom. Uses an approximation giving at least 5 decimal digits of precision when $n \geq 8$ or $n \leq 2$, and 3 decimal digits of precision when $3 \leq n \leq 7$ (see [21] and Figure L.28 of [10]).

`double Fdist_ChiSquare (double k, double x);`

Returns an approximation of $F(x)$, where F is the chi-square distribution with k degrees of freedom. Uses the approximation given in [27, p.116] for $n \leq 66$, and the normal approximation for $n > 66$.

`double Fdist_InvChiSquare1 (double k, double u);`

Returns a quick and dirty approximation of $F^{-1}(u)$, where F is the chi-square distribution with k degrees of freedom. Uses the approximation given in Figure L.24 of [10].

`double Fdist_InvChiSquare2 (double k, double u);`

Returns an approximation of $F^{-1}(u)$, where F is the chi-square distribution with k degrees of freedom. Uses the approximation given in Algorithm 91 of Applied Statistics (1975), 24, 3, and in Figure L.23 of [10].

Rand1

Ce module sert à générer des valeurs aléatoires selon différentes lois de probabilité. Des fonctions sont disponibles pour générer des valeurs selon les lois uniforme discrète, uniforme continue, exponentielle, Weibull, gamma, bêta, normale et Student. Elles utilisent toutes la méthode d'inversion [10, 30], sauf pour les lois gamma et bêta. L'usager peut aussi définir sa propre loi discrète, en donnant l'ensemble des valeurs qui peuvent être prises, et la fonction de répartition. Pour faire cela, on déclare une variable de type `Rand1_DiscreteDist`, on définit la distribution en question en appelant `Rand1_CreateDiscreteDist`, puis on utilise la fonction `Rand1_Discrete` pour générer des valeurs.

```
struct Rand1_record;
typedef struct Rand1_record * Rand1_DiscreteDist;
```

Une loi de probabilité discrète, que l'usager définira lui-même.

```
long Rand1_DiscreteUniform (Rand_Stream g, long N1, long N2);
```

Fournit une valeur aléatoire suivant la loi uniforme discrète sur l'ensemble des entiers $\{$`N1`$, \ldots,$ `N2`$\}$, en utilisant le générateur `g`. On doit avoir N2 $\geq$ N1 et la différence `N2 - N1` ne doit pas dépasser 2147483561.

```
Rand1_DiscreteDist Rand1_CreateDiscreteDist (unsigned long n,
                                  double V[], double F[]);
```

Retourne et permet de définir une loi de probabilité discrète. Le nombre de valeurs qui peuvent être prises est `n`, ces valeurs (distinctes) doivent être placées dans le tableau `V` en ordre croissant (dans `V[0..n-1]`), et le tableau `F` doit contenir la valeur de la fonction de répartition à chacune de ces valeurs. Ainsi, `F[i]` est la probabilité que la valeur de la variable aléatoire soit inférieure ou égale à `V[i]`. Les valeurs placées dans `F[0..n-1]` doivent être entre 0.0 et 1.0, en ordre croissant, et on doit avoir `F[n-1]`$= 1.0$.

```
double Rand1_Discrete (Rand_Stream g, Rand1_DiscreteDist D);
```

Fournit une valeur aléatoire suivant la loi discrète `D`, définie précédemment par l'usager, en utilisant le générateur `g`.

```
double Rand1_Uniform (Rand_Stream g, double A, double B);
```

Fournit une valeur aléatoire suivant la loi uniforme continue de paramètres `A` et `B`, en utilisant le générateur `g`. La valeur produite est comprise strictement entre `A` et `B`.

```
double Rand1_Normal (Rand_Stream g, double Mean, double StdDeviation);
```

Fournit une valeur aléatoire suivant la loi normale de moyenne $\mu =$ `Mean` et d'écart-type $\sigma =$ `StdDeviation`. Utilise l'inversion via la fonction `Rand1_InvNormalDist`.

```
double Rand1_InvNormalDist (double U);
```

Retourne $y = F^{-1}(u)$, où F est la fonction de répartition de la loi de normale de moyenne 0 et de variance 1. Utilise une approximation rationnelle donnant au moins 7 décimales de précision pour $10^{-20} <$`U`$< 1 - 10^{-20}$.

`double Rand1_Student (Rand_Stream g, long DegFreedom);`

Fournit une valeur aléatoire suivant la loi de Student avec `DegFreedom` degrés de liberté. Utilise l'inversion et une approximation rationnelle.

`double Rand1_InvStudentDist (long n, double u);`

Retourne $y = F^{-1}(u)$, où F est la fonction de répartition de la loi de Student à n degrés de liberté.

`double Rand1_Expon (Rand_Stream g, double Mean);`

Fournit une valeur aléatoire suivant la loi exponentielle de moyenne $\mu =$ `Mean`, c'est-à-dire, dont la fonction de répartition est

$$F(x) = 1 - e^{-x/\mu}, \qquad x > 0.$$

Appelle la fonction `Rand_RandU01` et utilise l'inversion.

`double Rand1_Weibull (Rand_Stream g, double Alpha, double Lambda);`

Fournit une valeur aléatoire suivant la loi Weibull de paramètres $\alpha =$ `Alpha` et $\lambda =$ `Lambda`, c'est-à-dire dont la fonction de répartition est

$$F(x) = 1 - e^{-\lambda x^\alpha}, \qquad x > 0.$$

On doit avoir $\alpha > 0$ et $\lambda > 0$. La moyenne est égale à

$$\frac{\Gamma(1 + 1/\alpha)}{\lambda^{1/\alpha}}, \qquad \alpha > 0.$$

Appelle la fonction `Rand_RandU01` et utilise l'inversion.

`double Rand1_Gamma (Rand_Stream g, double Alpha, double Lambda);`

Fournit une valeur aléatoire suivant la loi Gamma de paramètres $\alpha =$ `Alpha` et $\lambda =$ `Lambda`, c'est-à-dire dont la fonction de densité est

$$f(x) = \frac{\lambda^\alpha x^{\alpha-1} e^{-\lambda x}}{\Gamma(\alpha)}, \qquad x > 0.$$

On doit avoir $\alpha > 0$ et $\lambda > 0$. La moyenne et la variance sont α/λ et α/λ^2 respectivement. Utilise les fonctions RGS et RGKM3 des figures L.12 et L.13 de [10]. Il s'agit d'une méthode d'acceptation-rejet, de sorte que le nombre d'appels à la fonction `Rand_RandU01` est variable.

`double Rand1_Beta (Rand_Stream g, double A, double B);`

Fournit une valeur aléatoire suivant la loi Beta de paramètres $\alpha =$ `a` et $\beta =$ `b`, c'est-à-dire dont la fonction de densité est

$$f(x) = \frac{\Gamma(\alpha + \beta) x^{\alpha-1} (1 - x)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)}, \qquad 0 < x < 1.$$

On doit avoir $\alpha > 0$ et $\beta > 0$. Utilise l'algorithme de Cheng donné à la figure L.2 de [10]. Il s'agit d'une méthode d'acceptation-rejet, de sorte que le nombre d'appels à la fonction `Rand_RandU01` est variable.

Rand2

This module is an interface to the already-existing module `C--RAND` [44], written in C, which is an updated and improved version of a C-package developed by Kremer and Stadlober [29, 43].

The package had to be modified from its original version to provide access to the object-oriented approach of the `Rand` module in `SSC`. So the generator has been changed as well as non-portable functions implemented in `C--RAND`. The package has not been altered otherwise.

Description of the procedures have been taken directly from the documentation [44], with slight modifications.

Note: Procedures implemented in `C--RAND` have undefined behavior when parameters are out of range. That is, they may either halt, stall or provide wrong return values.

```
#include "Rand.h"

/* Prototypes for all user accessible C--Rand routines */

double bbbc (Rand_Stream gen, double a, double b);
```

Beta Distribution - Acceptance Rejection with log-logistic hats for the Beta Prime Distribution. Procedure `bbbc` samples a random number from the beta distribution with parameters `a,b` ($\texttt{a} > 0, \texttt{b} > 0$). Random number generator `gen` is used to provide randomness. Reference: R. C. H. Cheng [14].

```
double bsprc (Rand_Stream gen, double a, double b);
```

Beta Distribution - Stratified Rejection/Patchwork Rejection. Procedure `bsprc` samples a random variate from the beta distribution with parameters $\texttt{a} > 0$, $\texttt{b} > 0$. Reference: H. Sakasegawa [39], E. Stadlober and H. Zechner [45].

```
long bprsc (Rand_Stream gen, double n, double p);
```

Binomial Distribution - Patchwork Rejection/Inversion. Procedure `bprsc` samples a random number from the Binomial distribution with parameters `n` and `p` . It is valid for $\texttt{n}*min(\texttt{p}, 1-\texttt{p}) > 0$ where $\texttt{n} \geq 1$ and $0 < \texttt{p} < 1$. Note that parameter `n` must be of type `double` in this function. Reference: V. Kachitvichyanukul and B. W. Schmeiser [24].

```
long bruec (Rand_Stream gen, long n, double p);
```

Binomial Distribution - Ratio of Uniforms/Inversion. Procedure `bruec` samples a random number from the Binomial distribution as `bprsc`, but using a Ratio of Uniforms method combined with Inversion instead of a Patchwork of Rejection and Inversion method. Note that parameter `n` must be a `long` in this function. Reference: E. Stadlober [42].

```
long btpec (Rand_Stream gen, long n, double p);
```

Binomial-Distribution - Acceptance Rejection/Inversion. Procedure `btpec` combines an Acceptance Rejection method with Inversion for generating Binomial random numbers. Note that parameter `n` must be a `long` in this function. Reference: C. D. Kemp [26] and H. Zechner [47].

`double burr1 (Rand_Stream gen, double r, long nr);`

Burr II, VII, VIII, X Distributions - Inversion. Procedure `burr1` samples a random number from one of the Burr II, VII, VIII, X distributions with parameter `r` > 0 , where the number of the distribution is indicated by the variable `nr` (which must so be either 2, 7, 8 or 10 in order for the function to work properly). Reference: L. Devroye [17].

`double burr2 (Rand_Stream gen, double r, double k, long nr);`

Burr III, IV, V, VI, IX, XII Distribution - Inversion. Procedure `burr2` samples a random number from one of the Burr III, IV, V, VI, IX, XII distributions with parameter `r` > 0 and `k` > 0, where the number of the distribution is indicated by the variable `nr` (which must so be either 3, 4, 5, 6, 9 or 12 in order for the function to work properly). Reference: L. Devroye [17].

`double cin (Rand_Stream gen);`

Cauchy Distribution - Inversion. Procedure `cin` samples a random number from the standard Cauchy distribution $C(0, 1)$.

`double chru (Rand_Stream gen, double a);`

Chi Distribution - Ratio of Uniforms with shift. Procedure `chru` samples a random number from the Chi distribution with parameter `a` > 1. Reference: J. F. Monahan [37].

`double eln (Rand_Stream gen);`

Exponential Distribution - Inversion. Procedure `eln` samples a random number from the standard Exponential distribution $Exp(0, 1)$.

`double epd (Rand_Stream gen, double tau);`

Exponential Power Distribution - Acceptance Rejection. Procedure `epd` samples a random number from the Exponential Power distribution with parameter `tau` ≥ 1 by using the non-universal rejection method for logconcave densities. Reference: L. Devroye [17].

`double ev_i (Rand_Stream gen);`

Extreme value I Distribution - Inversion. Procedure `ev_i` samples a random number from the Extreme Value I distribution.

`double ev_ii (Rand_Stream gen, double a);`

Extreme Value II Distribution - Inversion. Procedure `ev_ii` samples a random number from the Extreme Value II distribution with parameter `a` > 0.

`double gds (Rand_Stream gen, double a);`

Gamma Distribution - Acceptance Rejection combined with Acceptance Complement. Procedure `gds` samples a random number from the standard gamma distribution with parameter `a` > 0. Acceptance Rejection algorithm is used for `a` < 1 and Acceptance Complement for `a` ≥ 1. Note: calls the `nsc` Standard Normal distribution function of this package. Reference: J. H. Ahrens and U. Dieter [5].

`double gll (Rand_Stream gen, double a);`

Gamma Distribution - Acceptance Rejection with log-logistic envelopes. Procedure `gll` produces a sample from the standard gamma distribution with parameter `a` > 0. Reference: R. C. H. Cheng [13].

`long gpoic (Rand_Stream gen, double my, double l);`

Generalized Poisson Distribution - Inversion/Ratio of Uniforms/Rejection. Procedure `gpoic` samples a random number from the Generalized Poisson distribution with the two parameters `my` > 0 and `lambda` < 1. Reference: P. Busswald [12] and L. Devroye [18].

`double gigru (Rand_Stream gen, double h, double b);`

Generalized Inverse Gaussian Distribution - Ratio of Uniforms. Procedure `gigru` samples a random number from the reparameterised Generalized Inverse Gaussian distribution with parameters `h` > 0 and `b` > 0 using Ratio of Uniforms method without shift for `h` ≤ 1 and `b` ≤ 1 and shift at mode `m` otherwise. Reference: J. S. Dagpunar [16] and K. Lehner [35].

`long geo (Rand_Stream gen, double p);`

Geometric Distribution - Inversion. Procedure `geo` samples a random number from the Geometric distribution with parameter $0 <$ `p` < 1.

`double hyplc (Rand_Stream gen, double a, double b);`

Hyperbolic Distribution - Non-Universal Rejection. Procedure `hyplc` samples a random number from the hyperbolic distribution with shape parameter `a` and `b` valid for `a` > 0 and $|$`b`$| <$ `a` using the non-universal rejection method for log-concave densities. Reference: L. Devroye [17].

`long h2pec (Rand_Stream gen, long N, long M, long n);`

Hypergeometric Distribution - Acceptance Rejection/Inversion. Procedure `h2pec` samples a random number from the Hypergeometric distribution with parameters `N` (number of red and black balls), `M` (number of red balls) and `n` (number of trials) valid for `N` ≥ 2, `M`, `n` $\leq$ `N`, `M` ≥ 0, `n` > 0. Note that parameters are of type `long`. Reference: V. Kachitvichyanukul and B. W. Schmeiser [23].

`long int hprsc (Rand_Stream gen, double N, double M, double n);`

Hypergeometric Distribution - Patchwork Rejection/Inversion. Procedure `hprsc` samples a random number from the Hypergeometric distribution as `h2pec` does, but using a different algorithm. Note that this time, parameters are of type `double` (but should be whole numbers for the expression to make sense.) Reference: H. Zechner [47].

`long hruec (Rand_Stream gen, long N, long M, long n);`

Hypergeometric Distribution - Ratio of Uniforms/Inversion. Procedure `hruec` uses another algorithm (Ratio of Uniforms combined with Inversion) to sample random numbers form an Hypergeometric distribution. Note that parameters are of type `long`. Reference: E. Stadlober [42].

`double john (Rand_Stream gen, double my, double sigma, long nr);`

Johnson Distribution - Transformation of one Normal random number. Procedure `john` samples a random number from the Johnson `S(B)`, `S(L)`, or `S(U)` distribution with parameters `my` and `sigma` > 0, where the type of the distribution is indicated by a pointer variable `nr`, which must be either 1, 2 or 3. Note: this procedure uses the `nsc` Standard Normal distribution function. Reference: N. L. Johnson [22].

`double lamin (Rand_Stream gen, double l3, double l4);`

Lambda Distribution - Inversion. Procedure `lamin` samples a random number from the Lambda distribution with parameter `l3` and `l4`. Reference: J. S. Ramberg and B. W. Schmeiser [38].

`double lapin (Rand_Stream gen);`

Laplace (Double Exponential) Distribution - Inversion. Procedure `lapin` samples a random number from the standard Laplace distribution $Lap(0, 1)$.

`double login (Rand_Stream gen);`

Logistic Distribution - Inversion. Procedure `login` samples a random number from the standard Logistic distribution $Log(0, 1)$.

`long lsk (Rand_Stream gen, double a);`

Logarithmic Distribution - Inversion/Transformation. Procedure `lsk` samples a random number from the Logarithmic distribution with parameter $0 <$ `p` < 1. Reference: A. W. Kemp [25].

`long nbp (Rand_Stream gen, double r, double p);`

Negative Binomial Distribution - Compound method. Procedure `nbp` samples a random number from the Negative Binomial distribution with parameters `r` (no. of failures given) and `p` (probability of success) valid for `r` > 0, $0 <$ `p` < 1. Note: this procedure uses both procedures `gds` and `pd` in its implementation. Reference: J. H. Ahrens and U. Dieter [1].

`double nsc (Rand_Stream gen);`

Normal Distribution - Sinus-Cosinus or Box/Muller Method. Procdeure `nsc` samples a random number from the standard Normal distribution $N(0, 1)$. Reference: G. E. P. Box and M. E. Muller [9].

`long pd (Rand_Stream gen, double my);`

Poisson Distribution - Tabulated Inversion combined with Acceptance Complement. Procedure `pd` samples a random number from the Poisson distribution with parameter `my` > 0. Tabulated Inversion for `my` < 10, Acceptance Complement for `my` ≥ 10. Uses `nsc` Standard Normal distribution random number generator in its implementation. Reference: J. H. Ahrens and U. Dieter [4].

`long pruec (Rand_Stream gen, double my);`

Poisson Distribution - Ratio of Uniforms/Inversion. Procedure `pruec` samples a random number from the Poisson distribution as function `pd` does, but using an Inversion method for `my` < 5.5 and a Ratio of Uniforms for `my` ≥ 5.5. Reference: E. Stadlober [42].

`long pprsc (Rand_Stream gen, double my);`

Poisson Distribution - Patchwork Rejection/Inversion. Procedure `pprsc` samples a random number from the Poisson distribution. For parameter `my` < 10, Tabulated Inversion is applied while for `my` ≥ 10 Patchwork Rejection is employed. Reference: H. Zechner [47].

`double slash (Rand_Stream gen);`

Slash Distribution - z/u (z from $N(0,1)$, u from $U(0,1)$). Procedure `slash` samples a random number from the Slash distribution. Note: makes use of the Standard Normal distribution provided by the `nsc` function.

`double trouo (Rand_Stream gen, double a);`

Student-t Distribution - Ratio of Uniforms. Procedure `trouo` samples a random number from the Student-t distribution with parameter `a` ≥ 1. Reference: A. J. Kinderman and J. F. Monahan [28].

`double tpol (Rand_Stream gen, double a);`

Student-t Distribution - Polar Method. Procedure `tpol` uses the polar method of Box/Muller for generating Normal variates in order to produce Student-t distributed random numbers, using parameter `a` > 0. Reference: R. W. Bailey [6].

`double stab (Rand_Stream gen, double alpha,  double delta);`

Stable Distribution - Integral Representations. Procedure `stab` samples a random number from the Stable distribution with parameters $0 <$ `alpha` ≤ 2 and $-1 \leq$ `delta` ≤ 1. Reference: J. M. Chambers et al. [20].

`double tra (Rand_Stream gen);`

Triangular Distribution - Inversion: $x = +-(1 - sqrt(u))$. Procedure `tra` samples a random number from the standard Triangular distribution in $(-1, 1)$.

`double mwc (Rand_Stream gen, double k);`

Von Mises Distribution - Acceptance Rejection. Procedure `mwc` samples a random number from the von Mises distribution ($-\pi \leq x \leq \pi$) with parameter `k` > 0 via rejection from the wrapped Cauchy distibution. Reference: D. J. Best and N. I. Fisher [7].

`double win (Rand_Stream gen, double c);`

Weibull Distribution - Inversion. Procedure `win` samples a random number from the Weibull distribution with parameter `c` > 0.

`long zeta (Rand_Stream gen, double ro, double pk);`

Zeta Distribution - Acceptance Rejection. Procedure `zeta` samples a random number from the Zeta distribution with parameters `ro` > 0 and `pk` ≥ 0. Reference: J. Dagpunar [15].

Randlib

This module is an interface to the already-existing module `RANDLIB` [11], written in C, available at `http://odin.mdacc.tmc.edu/anonftp/page_2.html#RANDLIB`. The original module has been slightly modified from its original version to provide full compatibility with the `Rand` module. Otherwise, all of the functions have been kept without any change. All methods that generate a random deviate ask for a `Rand_Stream`, that must has been created previously.

```
#include "Rand.h"

/* Prototypes for all user accessible Randlib routines */

double genbet (Rand_Stream gen, double a, double b);
```

Returns a single random deviate from the beta distribution with parameters `a` and `b`. The density of the beta is $x^{(\mathtt{a}-1)} * (1-x)^{(\mathtt{b}-1)}/B(\mathtt{a},\mathtt{b})$ for $0 < \mathrm{x} < 1$. Method used: R. C. H. Cheng [14].

```
double genchi (Rand_Stream gen, double DF);
```

Returns a single random deviate from the distribution of a chisquare with `DF` degrees of freedom random variable. `DF` must be > 0.

```
double genexp (Rand_Stream gen, double AV);
```

Returns a single random deviate from an exponential distribution with mean `AV`, which must be ≥ 0. For details about algorithm, see J. H. Ahrens and U. Dieter [2].

```
double genf (Rand_Stream gen, double DFN, double DFD);
```

Returns a single random deviate from the F (variance ratio) distribution with `DFN` degrees of freedom in the numerator and `DFD` degrees of freedom in the denominator. `DFN` and `DFD` must be > 0.

```
double gengam (Rand_Stream gen, double A, double R);
```

Returns a single random deviate from the gamma distribution with location parameter `A` and shape parameter `R`. `A` and `R` must be > 0. Uses methods from Ahrens and Dieter [5, 1].

```
void genmn (Rand_Stream gen, double *Parm, double *X, double *Work);
```

Generates a multivariate normal random deviate. Return in `X` the generated vector deviate. In `Parm` are the parameters needed. Parm must be set by a previous call to `setgmn`. `Work` is a scratch array.

```
void setgmn (double *Meanv, double *Covm, long P, double *Parm);
```

Returns in `Parm` the needed information to call `genmn`, i.e. `P`, `Meanv` and the Cholesky factorization of `Covm`. `Meanv` is the mean vector of multivariate normal distribution. `Covm` is the covariance matrix of the multivariate of normal distribution and is returned destroyed. `P` is the dimension of the normal, or length of `Meanv`. `Parm` dimension must be at least `P*(P+3)/2 + 1`.

`void genmul (Rand_Stream gen, long N, double *P, long NCAT, long *IX);`

Generates an observation from the multinomial distribution. `N` is the number of events that will be classified into one of the categories 1..`NCAT` (`N` ≥ 0). `P` is the vector of probabilities. `P`(i) is the probability that an event will be classified into category i. Thus, `P`(i)must be [0,1]. Only the first `NCAT`-1 `P`(i) must be defined since `P`(`NCAT`) is 1.0 minus the sum of the first `NCAT`-1 `P`(i). `NCAT` is the number of categories. Length of `P` and IX. (`NCAT` > 1). Result will be put in IX. All IX(i), the number of events of the ith category that occured, will be nonnegative and their sum will be `N`. Uses algorithm from Devroye [17].

`double gennch (Rand_Stream gen, double DF, double Nonc);`

Returns a single random deviate from the distribution of a noncentral chisquare with `DF` degrees of freedom and noncentrality parameter `Nonc`. `DF` must be ≥ 1.0 and `Nonc`, ≥ 0.

`double gennf (Rand_Stream gen, double DFN, double DFD, double Nonc);`

Returns a random deviate from the noncentral F (variance ratio) distribution with `DFN` degrees of freedom in the numerator, and `DFD` degrees of freedom in the denominator, and noncentrality parameter `Nonc`. `DFN` must be ≥ 1.0, `DFD` must be > 0 and `Nonc` must be ≥ 0.

`double gennor (Rand_Stream gen, double AV, double SD);`

Returns a single random deviate from a normal distribution with mean, `AV`, and standard deviation, `SD`. `SD` must be ≥ 0. Method: Ahrens and Dieter [3].

`void genprm (Rand_Stream gen, long *Iarray, long LArray);`

Generates a random permutation of the elements of `IArray`. `LArray` is the length of `IArray`.

`long ignbin (Rand_Stream gen, long N, double P);`

Returns a single random deviate from a binomial distribution whose number of trials is `N` and whose probability of an event in each trial is `P`. Method used: algorithm BTPE from Kachitvichyanukul and Schmeiser [24].

`long ignnbn (Rand_Stream gen, long N, double P);`

Returns a single random deviate from a negative binomial distribution. `N` is the number of events wanted and `P` is the probability of an event in each trial. Algorithm from Luc Devroye [17].

`long ignpoi (Rand_Stream gen, double AV);`

Returns a single random deviate from a Poisson distribution with mean `AV`. `AV` must be > 0.0. For details about algorithm, see Ahrens and Dieter [4].

`long ignuin (Rand_Stream gen, long Low, long High);`

Returns a single random deviate uniformly distributed between `Low` and `High`. (`High - Low`) must be $< 2{,}147{,}483{,}561$.

Sim

Ce module fait l'interface avec l'exécutif du progiciel de simulation. Il fournit des procédures permettant d'initialiser, de démarrer ou d'arrêter la simulation, de même qu'une fonction qui retourne la valeur de l'horloge.

`void Sim_Init (void);`

Réinitialise la simulation en vidant la liste des événements, si elle ne l'est pas déjà, et en remettant l'horloge à zéro.

`void Sim_Start (void);`

Démarre l'exécutif de la simulation. Lors de son appel, il doit déjà y avoir au moins un événement ou un processus de prévu.

`void Sim_Stop (void);`

Arrête l'exécutif de la simulation et retourne le contrôle au programme principal (ou à l'une de ses procédures), juste après l'instruction `Sim_Start` qui a démarré cette simulation. A noter cependant que l'exécutif ne s'arrêtera que lorsqu'on lui aura tranféré le contrôle.

`double Sim_Time (void);`

Retourne la valeur de l'instant courant de la simulation.

Event

Ce module offre les outils nécessaires pour effectuer une simulation avec vision par événements. Le type `Event_Type`, prédéfini dans ce module, correspond à un type d'événement qui est associé à une procédure sans paramètre. Pour chaque type d'événement, l'usager doit déclarer une variable de type `Event_Type`, puis appeler `Event_Create` pour lui associer sa procédure correspondante, qui sera exécutée à chaque occurence d'un événement de ce type. On peut prévoir un événement dans un délai donné, par la procédure `Event_Schedule`. On donne alors le type de l'événement, le délai, et un pointeur sur les attributs (paramètres) associés à cet événement. On peut récupérer la valeur de ce pointeur, à l'intérieur de la procédure qui exécute l'événement en question, en appelant la fonction `Event_Attrib`. La procédure `Event_Cancel` permet d'annuler un événement déjà prévu, selon un attribut particulier. Pour chaque événement prévu, le système crée un avis d'événement (`Event_Notice`) qui contient le type de l'événement (le `Event_Type`), son instant d'occurence (supérieur ou égal à l'instant courant) et un pointeur sur ses attributs, s'il en a.

Lorsqu'on veut prévoir un événement pour l'instant courant, et que l'on veut qu'il soit placé au tout début de la liste des événements, avant les autres déjà prévus pour le même instant s'il y a lieu, on appelle plutôt `Event_ScheduleNext` au lieu de `Event_Schedule`. Les procédures `Event_ScheduleNotice` et `Event_ScheduleNoticeNext` permettent de prévoir un événement tout en récupérant le pointeur sur l'avis d'événement. Cela peut permettre par exemple d'annuler ultérieurement cet avis d'événement en appelant `Event_CancelNotice`. On peut aussi placer un avis d'événement juste avant ou juste après un autre avis qui se trouve déjà dans la liste (voir `Event_ScheduleNoticeBefore` et `Event_ScheduleNoticeAfter`). D'autres procédures permettent d'obtenir le type, le pointeur sur les attributs, et l'instant d'occurence d'un événement prévu, de retrouver un avis d'événement satisfaisant une condition particulière, ou de faire imprimer la liste des événements prévus.

```
typedef void (*Event_Procedure) (void);
```

Un type pour les procédures.

```
struct Event_RecordType;
typedef struct Event_RecordType * Event_Type;
```

Un type d'événement. Pour chaque type d'événement qu'il veut utiliser, l'usager doit déclarer une variable de ce type, puis appeler la procédure `Event_Create`.

```
struct Event_InfoNotice;
typedef struct Event_InfoNotice * Event_Notice;
```

Un avis d'événement.

```
typedef int (*Event_CondProc) (void *);
```

Type de procédure utilisé comme paramètre dans la procédure `Event_FindNoticeSuchThat`.

```
Event_Type Event_Create (Event_Procedure P, char * Name);
```

Associe la procédure `P` et l'identificateur `Name` a un type d'événement qui est retourné par la fonction. Cette procédure doit être obligatoirement appelée pour chaque type d'événement. `Name` servira à identifier ce type d'événement si on fait imprimer la liste des événements.

```
void Event_Schedule (Event_Type E, double D, void * Par);
```

Prévoit un événement de type `E`, en insérant un avis dans la liste des événements. Son occurence est prévue pour l'instant "`Sim_Time() + D`", c'est-à-dire dans `D` unités de temps. L'usager peut associer à cet événement un bloc de paramètres (un attribut), sous forme d'un pointeur dont la valeur est donnée au paramètre formel `Par`. Il pourra récupérer la valeur de ce pointeur lors de l'exécution de l'événement, en utilisant la fonction `Event_Attrib`. `D` doit être non négatif, sinon une erreur est signalée et rien d'autre n'est effectué. Lorsque deux ou plusieurs événements sont prévus pour le même instant, ils sont placés dans la liste (et exécutés) dans l'ordre selon lequel ils ont été prévus.

```
void Event_ScheduleNext (Event_Type E, void * Par);
```

Tout comme `Event_Schedule`, sauf que l'événement prévu est placé au tout début de la liste d'événements, et doit s'exécuter à l'instant courant. Si d'autres événements sont aussi prévus pour l'instant courant, l'événement que l'on prévoit maintenant s'exécutera avant eux.

```
void Event_Cancel (Event_Type E, void * Par);
```

Annule (enlève de la liste d'événements) le premier avis d'événement de type `E` et qui a été prévu avec la valeur `Par` pour ses attributs. La recherche s'effectue à partir du début de la liste des événements (par ordre chronologique). Si la liste d'événements est vide, ou que l'événement recherché est inexistant, rien n'est effectué.

```
void * Event_Attrib (void);
```

Retourne le pointeur sur les attributs de l'événement en cours d'exécution. Ce pointeur est la valeur du paramètre `Par` fourni lors de sa prévision. C'est à l'usager de s'assurer que ce résultat est affecté à une variable du même type (c'est-à-dire un pointeur sur le même type de structure) que la variable passée en paramètre lors de la prévision de cet événement. Attention : si l'usager appelle cette procédure ailleurs que dans une procédure exécutant un événement, les résultats sont imprévisibles (il est possible que l'on soit en train d'exécuter un événement défini par SSC...).

```
Event_Notice Event_ScheduleNotice (Event_Type E, double D, void * Par);
```

Tout comme `Event_Schedule`, mais retourne l'avis d'événement associé.

```
Event_Notice Event_ScheduleNoticeNext (Event_Type E, void * Par);
```

Tout comme `Event_ScheduleNext`, mais retourne l'avis d'événement associé.

```
Event_Notice Event_ScheduleNoticeBefore (Event_Type E, void * Par,
                                         Event_Notice N1);
```

Tout comme `Event_ScheduleNotice`, sauf que le nouvel avis d'événement est placé immédiatement avant l'avis `N1` dans la liste. `N1` doit être un avis d'événement déjà prévu. Cette procédure peut

être utile par exemple pour placer un événement “espion” qui s’exécutera juste avant un autre événement donné.

```
Event_Notice Event_ScheduleNoticeAfter (Event_Type E, void * Par,
                                        Event_Notice N1);
```

Tout comme `Event_ScheduleNoticeBefore`, sauf que le nouvel avis d’événement est placé immédiatement après l’avis `N1` dans la liste.

```
Event_Notice Event_FindNoticeSuchThat (Event_Type E, Event_CondProc Cond);
```

Retourne l’avis d’événement associé au premier événement de type `E` dans la liste tel que la procédure `Cond`, appliquée au pointeur sur les attributs de cet événement, retourne `1`. S’il n’y en a pas, cette procédure retourne `NULL`.

```
void Event_CancelNotice (Event_Notice * N);
```

Annule l’événement associé à l’avis d’événement `N`.

```
Event_Type Event_NoticeType (Event_Notice N);
```

Retourne le type de l’événement associé à l’avis d’événement `N`.

```
void * Event_NoticeAttrib (Event_Notice N);
```

Retourne le pointeur sur les attributs de l’événement associé à `N`. Ce pointeur est la valeur du paramètre `Par` fourni lors de sa prévision.

```
double Event_NoticeTime (Event_Notice N);
```

Retourne l’instant (prévu) d’occurence de l’événement associé à `N`.

```
void Event_PrintEventList (void);
```

Imprime la liste des événements dans le fichier de sortie courant.

```
void Event_EmptyEventList (void);
```

Vide complètement la liste des événements prévus, c’est-à-dire annule tous ces événements.

```
void Event_DeleteType (Event_Type * E);
```

Détruit le type d’événement `E` et récupère l’espace mémoire utilisé pour sa description.

Cont

Ce module permet la simulation continue, pour laquelle l'évolution de certaines variables en fonction du temps est régie par des équations différentielles. Dans un même modèle, on peut mélanger des événements discrets, des processus, et des variables "continues" qui obéissent à des équations différentielles. Il doit cependant y avoir un nombre fini de telles variables, et pour chacune d'entre elles, on doit donner l'équation différentielle qui la régit, sous forme d'une fonction. Ce module ne permet pas de traiter les équations aux dérivées partielles, que l'on simule habituellement par des techniques d'éléments finis.

Pour chaque variable continue que l'on veut utiliser, on doit déclarer une variable de type `Cont_Var`, puis la créer en appelant `Cont_Create`. Lors de la création de la variable, on doit donner sa valeur initiale, et une fonction qui retourne sa dérivée en fonction du temps. L'évolution de la valeur de la variable se fait en intégrant cette fonction. Pour chaque variable, on peut démarrer l'intégration en appelant `Cont_StartInteg`, et la stopper en appelant `Cont_StopInteg`. Lors du démarrage, on peut aussi choisir un type d'événement qui sera exécuté à chaque pas d'intégration pour cette variable (juste après avoir effectué ce pas). Un tel événement pourrait par exemple vérifier si une variable continue a atteint un certain seuil, ou encore mettre à jour un graphique illustrant l'évolution du système. Avant de démarrer toute intégration, on doit appeler `Cont_SelectMethod` pour choisir la méthode et la longueur h du pas d'intégration. Le même h et la même méthode seront utilisés pour toutes les variables. Le pas d'intégration qui fait passer la variable de la valeur qu'elle a à l'instant $t-h$, à celle qu'elle doit avoir à l'instant t, se fait lorsque l'horloge de la simulation atteint l'instant t.

Lors de la création d'une variable, on peut aussi lui associer un paramètre (en général un pointeur), que l'on pourra ensuite récupérer en appelant `Cont_VarAttrib`. La fonction `Cont_CurrentVar`, lorsqu'appelée à l'intérieur de la fonction de type `Cont_Deriv` associée à une variable continue `V`, durant l'intégration, retourne `V`. En particulier, une combinaison de ces deux fonctions permet d'examiner le paramètre de la variable que l'on est en train d'intégrer. On peut réinitialiser la valeur d'une variable continue en appelant `Cont_Init`, et examiner sa valeur courante en appelant `Cont_Value`. Enfin, pour la détruire et récupérer l'espace qu'elle occupait, il suffit d'appeler `Cont_Delete`.

```
struct Cont_InfoVar;
typedef struct Cont_InfoVar * Cont_Var;
```

Pour chaque variable continue que l'on veut utiliser, on doit déclarer une variable de ce type, puis appeler `Cont_Create` pour la créer.

```
typedef double (*Cont_Deriv) (double);
```

Type de procédure à passer en paramètre dans `Cont_Create`.

```
typedef enum {Cont_Euler, Cont_RungeKutta2, Cont_RungeKutta4} Cont_Method;
```

Une méthode d'intégration. `Cont_Euler`: la méthode d'Euler; `Cont_RungeKutta2`: la méthode d'Euler modifiée (Runge Kutta d'ordre 2); `Cont_RungeKutta4`: Runge Kutta d'ordre 4.

`Cont_Var Cont_Create (double Val, Cont_Deriv D, void * Par);`

Crée et retourne une variable continue , l'initialise à la valeur `Val`, et lui associe la procédure `D`. Cette procédure, appelée avec le paramètre `t`, doit retourner la dérivée de la valeur de la variable en fonction du temps, au temps `t`, en fonction des valeurs courantes des variables (continues ou autres) du modèle. Le paramètre `Par` est en général un pointeur sur un bloc d'information quelconque associé à cette variable (ou toute autre variable compatible avec `void *`). On pourra récupérer sa valeur en appelant `Cont_VarAttrib`.

`void Cont_Init (Cont_Var V, double Val);`

Réinitialise la valeur de la variable `V` à `Val`.

`double Cont_Value (Cont_Var V);`

Retourne la valeur courante de la variable `V`.

`void * Cont_VarAttrib (Cont_Var V);`

Retourne la valeur du paramètre associé à `V`, c'est-à-dire la valeur de `Par` fournie lors de sa création.

`Cont_Var Cont_CurrentVar (void);`

Lorsqu'appelée à l'intérieur d'une procédure de type `Cont_Deriv` associée à une variable `V`, durant l'intégration, cette fonction retourne `V`. Sinon, retourne `NULL`.

`void Cont_SelectMethod (Cont_Method M, double h);`

Choisit la méthode d'intégration qui fait évoluer l'état des variables continues, et la longueur du pas d'intégration. Pour toutes les variables pour lesquelles l'intégration a été démarrée, la méthode d'intégration sera `M`, avec un pas d'intégration de `h` unités de temps.

`void Cont_StartInteg (Cont_Var V, Event_Type E, void * Par);`

Démarre le processus d'intégration qui fait évoluer l'état de la variable `V`. Juste après chaque pas d'intégration, un événement de type `E`, avec `Par` comme pointeur sur ses paramètres, sera exécuté. Si on ne veut pas d'un tel événement, on passe `Event_Type (NULL)` pour valeur de `E`.

`void Cont_StopInteg (Cont_Var V);`

Stoppe l'intégration qui avait été démarrée par `Cont_StartInteg` pour la variable `V`. La valeur de `V` demeurera celle calculée lors du dernier pas d'intégration effectué avant l'appel à `Cont_StopInteg`.

`void Cont_Delete (Cont_Var * V);`

Détruit la variable continue `V`, et récupère l'espace qu'elle occupait.

Stat

Le module `Stat` permet de recueillir des statistiques, d'obtenir des rapports et de calculer des intervalles de confiance. Il fournit un type prédéfini appelé `Stat_Block`, qui correspond à un bloc d'information pour le recueil de statistiques. Pour chaque quantité pour laquelle on désire recueillir des statistiques, on déclare une variable de type `Stat_Block`, puis on crée le bloc d'information correspondant en appelant `Stat_Create`. On doit distinguer deux types de blocs statistiques. On a un bloc de type `Stat_Tally` lorsqu'on s'intéresse à une séquence d'observations (de valeurs numériques) $X_1, X_2, X_3, \ldots$ d'une variable. Par ailleurs, lorsqu'on s'intéresse à l'évolution dans le temps de la valeur d'une variable, $X(t), t \geq 0$, on a un bloc de type `Stat_Accumulate`. Par exemple, pour estimer la longueur moyenne d'une file d'attente en fonction du temps, on utilisera un bloc de type `Stat_Accumulate`, tandis que pour estimer la durée moyenne d'attente par client, on utilisera un bloc de type `Stat_Tally`. Le type d'un bloc doit être déclaré lors de sa création.

Dans *SSC*, les blocs statistiques déclarés par l'usager ne sont pas mis à jour automatiquement par le système. L'usager doit appeler la procédure `Stat_Update` pour fournir la nouvelle valeur de X_i chaque fois qu'une nouvelle valeur est observée, dans le cas d'un bloc de type `Stat_Tally`, et pour donner la nouvelle valeur de $X(t)$ chaque fois que la valeur de la variable est modifiée, dans le cas d'un bloc de type `Stat_Accumulate`.

La procédure `Stat_Init` permet de réinitialiser, à n'importe quel moment, tous les compteurs associés à un bloc statistique. On peut donc réinitialiser sélectivement ces derniers au besoin. Lorsqu'on effectue plusieurs répétitions, par exemple, on peut utiliser un bloc statistique pour recueillir la distribution des moyennes des différentes répétitions, et réinitialiser seulement les autres blocs au début de chaque répétition. `Stat_Delete` permet d'éliminer un bloc statistique et d'en récupérer l'espace.

La procédure `Stat_Report` permet d'obtenir un rapport complet sur un bloc statistique. Lorsqu'on veut connaître seulement la valeur d'une statistique particulière, on peut faire appel aux fonctions `Stat_NumberObs`, `Stat_Minimum`, `Stat_Maximum`, `Stat_Sum`, `Stat_Average`, `Stat_Variance` et `Stat_StandardDev`. A noter cependant que le nombre d'observations, la variance et l'écart-type ne sont disponibles que pour les blocs de type `Stat_Tally`. La variance et l'écart-type n'ont d'utilité, en général, que lorsque les valeurs observées sont indépendantes (par exemple les moyennes de plusieurs répétitions). On peut aussi, sous ces mêmes restrictions, calculer un intervalle de confiance, de niveau spécifié, pour la moyenne d'une variable. Il suffit d'appeler la procédure `Stat_ConfidenceInterval` ou `Stat_ReportConfidenceInterval`. Ces procédures supposent cependant que les valeurs observées sont indépendantes et identiquement distribuées selon une loi normale. Les intervalles de confiance obtenus ne seront valides que si cette hypothèse est satisfaite, ou n'est pas trop loin de l'être. Pour un bloc de type `Stat_Accumulate`, tout appel à l'une des procédures mentionnées dans ce paragraphe provoque une mise-à-jour automatique (par un appel à `Stat_Update`).

```
struct Stat_InfoBlock;
typedef struct Stat_InfoBlock * Stat_Block;
```

Bloc statistique renfermant toutes les informations nécessaires pour le recueil des statistiques concernant une variable donnée. Chaque bloc statistique que l'on veut utiliser doit être identifié par une variable de type `Stat_Block`, dont l'identificateur servira de nom au bloc statistique en question, puis créé à l'aide de la procédure `Stat_Create`.

```
typedef enum {Stat_Tally, Stat_Accumulate} Stat_BlockType;
```

Sorte de bloc statistique.

```
Stat_Block Stat_Create (Stat_BlockType Kind, char * Name);
```

Crée et retourne un bloc statistique, dont le type est spécifié par `Kind`. Alloue l'espace et initialise tous les compteurs associés à ce bloc. Le paramètre `Name` permet de donner un nom au block sous forme d'une chaîne de caractères. Ce nom sera utilisé lorsqu'on fera imprimer un rapport pour ce block.

```
void Stat_Init (Stat_Block X);
```

Initialise ou réinitialise tous les compteurs associés au bloc statistique `X`. Le bloc `X` doit avoir été créé auparavant. A noter que dans le cas où `X` est de type `Stat_Accumulate`, `Stat_Init` ne modifie pas la valeur courante de la variable. Cette valeur courante est toujours celle du dernier appel à `Stat_Update`. Pour l'initialiser, la modifier, ou la remettre à 0, il faut appeler `Stat_Update`. Habituellement, un appel à `Stat_Init` pour un bloc de type `Stat_Accumulate` devrait être immédiatement suivi d'un appel à `Stat_Update`.

```
void Stat_Update (Stat_Block X, double V);
```

Met à jour tous les compteurs associés au bloc statistique `X`. Si `X` est de type `Stat_Tally`, une nouvelle observation de la variable, de valeur `V`, est ajoutée. Si `X` est de type `Stat_Accumulate`, cette mise a jour indique qu'à partir de l'instant courant, la variable associée au bloc `X` prend la valeur `V`.

```
void Stat_Report (Stat_Block X);
```

Imprime, dans le fichier de sortie (habituellement à l'écran), un rapport complet sur la variable associée au bloc statistique `X`, depuis la dernière initialisation de ce bloc.

```
unsigned long Stat_NumberObs (Stat_Block X);
```

Pour une statistique de type `Stat_Tally`, fournit le nombre de valeurs observées pour la variable associée au bloc statitique `X` depuis la dernière initialisation de ce bloc. Pour une statistique de type `Stat_Accumulate`, imprime un message d'erreur.

```
double Stat_Minimum (Stat_Block X);
```

Fournit la plus petite valeur qu'a prise la variable associée au bloc statistique `X` depuis la dernière initialisation de ce bloc.

```
double Stat_Maximum (Stat_Block X);
```

Fournit la plus grande valeur qu'a prise la variable associée au bloc statistique `X` depuis la dernière initialisation de ce bloc.

```
double Stat_Sum (Stat_Block X);
```

Pour une statistique de type `Stat_Tally`, fournit la somme des valeurs prises par la variable associée au bloc `X` depuis sa dernière initialisation. Pour une statistique de type `Stat_Accumulate`, fournit l'intégrale de la valeur de la variable, par rapport au temps, depuis la dernière initialisation du bloc `X` jusqu'à l'instant courant.

```
double Stat_Average (Stat_Block X);
```

Pour une statistique de type `Stat_Tally`, fournit la moyenne des observations de la variable associée au bloc `X` (somme des valeurs divisée par le nombre d'observations) depuis sa dernière initialisation. Pour une statistique de type `Stat_Accumulate`, fournit la valeur moyenne de la variable, par unité de temps, depuis la dernière initialisation du bloc.

```
double Stat_Variance (Stat_Block X);
```

Pour une statistique de type `Stat_Tally`, fournit la variance des observations de la variable associée au bloc `X` (somme des carrés des écarts à la moyenne, divisée par le nombre d'observations moins 1) depuis sa dernière initialisation. Si le nombre d'observations est inférieur à 2, ou si `X` est de type `Stat_Accumulate`, un message d'erreur sera imprimé. *Attention*: L'implantation utilise la double précision, mais il peut y avoir une erreur numérique significative lorsque la variance est *très* petite par rapport à la moyenne.

```
double Stat_StandardDev (Stat_Block X);
```

Pour une statistique de type `Stat_Tally`, fournit l'écart-type des observations. Appelle la fonction `Stat_Variance` et extrait la racine carrée.

```
void Stat_ConfidenceInterval (Stat_Block X, double Level, double * Center,
                              double * Radius);
```

Pour une statistique de type `Stat_Tally`, fournit le point milieu et le rayon d'un intervalle de confiance, de niveau spécifié par l'usager, pour la vraie moyenne μ de la variable associée à `X`. Cet intervalle est calculé en utilisant la statistique :

$$T = \frac{\overline{X} - \mu}{S_x / \sqrt{n}}$$

où n est le nombre d'observations de la variable, $\overline{X}$ = `Stat_Average(X)` est la moyenne échantillonnale, et S_x = `Stat_StandardDev(X)` est l'écart-type échantillonnal. Si on suppose que les observations de la variable sont indépendantes et identiquement distribuées suivant une loi normale, alors la statistique T suit la loi de Student à $n-1$ degrés de liberté. Lorsque cette hypothèse n'est pas vérifiée, les valeurs fournies peuvent avoir une valeur statistique doûteuse. Si $n < 2$, ou si la statistique `X` est de type `Stat_Accumulate`, un message d'erreur sera imprimé.

```
void Stat_ReportConfidenceInterval (Stat_Block X, double Level);
```

Pour une statistique de type `Stat_Tally`, imprime un intervalle de confiance, de niveau spécifié par l'usager, pour la vraie moyenne μ de la variable associée à `X`. Cet intervalle est calculé en utilisant la procédure `Stat_ConfidenceInterval`.

```
void Stat_Delete (Stat_Block * X);
```

Détruit le bloc statistique `X` et récupère l'espace mémoire qu'il utilisait.

List

`List` est un module de gestion de listes doublement chaînées. Le type `List_List` est prédéfini dans le module. Un objet de ce type est constitué d'une tête de liste, contenant l'information générale sur cette liste, et peut contenir zéro ou plusieurs objets.

Ce module gère chaque liste sans connaître le type des objets qui s'y trouvent. Les "objets" insérés ou retirés des listes sont de type `void *`, et `List` ne manipule que ces pointeurs. On peut construire des listes contenant à peu près n'importe quoi, comme par exemple des processus, des blocs statistiques, des ressources, ou même d'autres listes. Il peut y avoir plusieurs types d'objets dans une même liste, et un objet peut se trouver dans plusieurs listes à la fois. En pratique, les types des objets d'une liste sont déclarés dans le programme de l'usager et sont en général des pointeurs sur des structures de données de types biens spécifiques. L'usager doit donc être prudent, et s'assurer que chaque fois qu'un objet est retiré d'une liste, il est placé dans une variable du même type (c'est-à-dire le même type de pointeur) que celle utilisée lors de son insertion dans la liste.

Pour chaque liste qu'il veut utiliser, l'usager doit déclarer une variable de type `List_List`, puis créer cette liste. Une liste peut être ordonnée ou bien selon la façon dont les objets y sont insérés, ou encore selon une procédure d'ordonnancement des objets fournie par l'usager lors de sa création. Le choix de l'un ou l'autre de ces deux modes d'ordonnancement est fixé une fois pour toutes lors de la création de la liste, et dépend de la procédure appelée pour effectuer cette création: `List_Create` et `List_CreateOrd` respectivement. Dans le premier cas, on dit (par un léger abus de langage) qu'on a une liste de type *non ordonné*, alors que dans le second cas, on dit que la liste est de type *ordonné*.

Pour insérer des objets dans une liste, on utilise l'une des procédures `List_Insert` ou `List_InsertOrd`, selon le type de la liste. La fonction `List_Size` permet de connaître le nombre d'objets qui se trouvent dans une liste, et la fonction `List_Belongs` permet de savoir si un objet particulier s'y trouve.

Pour chaque liste, le système mémorise la position du dernier objet référencé ou inséré. Cet objet s'appelle l'objet courant. Les procédures `List_Remove` et `List_View` permettent de retirer un objet d'une liste, ou de consulter (observer) un objet d'une liste sans le retirer de la liste. On peut retirer ou consulter l'objet courant, son successeur ou son prédécesseur, de même que le premier ou le dernier objet de la liste. Cela permet, par exemple, de parcourir une liste pour y trouver un objet ayant une propriété particulière. On peut aussi, lorsqu'on connaît le nom d'un objet (c'est-à-dire la valeur du pointeur associé), utiliser la procédure `List_RemoveObject` pour le retirer d'une liste.

D'autres procédures permettent d'unir deux listes, de trier une liste, ou de la diviser en deux. `List_Concatenate` met bout-à-bout deux listes non ordonnées pour en faire une seule liste, tandis que `List_Merge` fusionne en une seule liste deux listes ordonnées selon la même fonction d'ordonnacement. Une liste ordonnée peut aussi être réordonnée, par la procédure `List_Sort`, selon une nouvelle fonction fournie par l'usager. Enfin, il est aussi possible, par la procédure `List_Split`, de diviser une liste en deux nouvelles listes au niveau de l'objet courant.

SSC recueille des statistiques automatiquement sur les listes, mais seulement si on le lui demande (cela permet d'accélérer l'exécution dans les cas où ce n'est pas nécessaire). Pour chaque liste pour laquelle on le désire, il suffit d'appeler la procédure `List_CollectStat`, et le système lui associe deux blocs statistiques, mis-à-jour automatiquement. L'un de ces blocs est de type `Stat_Accumulate`, et sert à mesurer l'évolution de la taille de la liste en fonction du temps, tandis que l'autre est de type `Stat_Tally`, et échantillonne les durées de séjour des objets dans la liste. Les fonctions `List_StatSize` et `List_StatSojourn` retournent ces blocs statistiques, ce qui permet d'appeler les fonctions du module `Stat` pour examiner certaines statistiques particulières. La procédure `List_Report` fournit par ailleurs un rapport statistique complet sur une liste donnée, et `List_InitStat` permet de réinitialiser les deux blocs statistiques associés à une liste.

La procédure `List_Init` vide une liste de tous ses objets et réinitialise sa tête (y compris les blocs statistiques associés, s'il y a lieu), et `List_Delete` détruit une liste et récupère l'espace occupé par la tête de liste. A noter cependant que ces procédures ne détruisent pas les objets créés et gérés par l'usager, c'est-à-dire ceux qui correspondent aux pointeurs qui se trouvent dans la liste créée ou détruite. L'usager peut donc réutiliser ces objets à d'autres fins. En général, lorsqu'on effectue plusieurs répétitions d'une simulation, on doit vider la plupart des listes après chaque répétition. On peut utiliser `List_Init` si les objets qui s'y trouvent n'ont pas à être détruits parce qu'ils peuvent être réutilisés d'une répétition à l'autre. Lorsque les objets qui s'y trouvent doivent être détruits pour en récupérer l'espace, l'usager doit alors les retirer de la liste un à un pour les détruire.

```
typedef struct List_Head * List_List;
```

Pour chaque liste qu'il veut utiliser, l'usager doit déclarer une variable de type `List_List`, puis créer cette liste.

```
typedef enum {List_First, List_Last, List_Previous,
      List_Next, List_Current} List_Position;
```

Type de position où l'on peut observer, retirer ou insérer un objet dans une liste. Note: l'insertion ne peut être faite à la position `List_Current`.

```
typedef int (*List_OrdProc) (void *, void *);
```

Type de procedure utilisé comme paramètre dans les procédures `List_CreateOrd` et `List_Sort`.

```
typedef int (*List_CondProc) (void *);
```

Type de procedure utilisé comme paramètre dans la procédure `List_FindFirstSuchThat`.

```
typedef void (*List_ActionProc) (void *);
```

Type de procedure utilisé comme paramètre dans la procédure `List_ForEachInListDo`.

```
List_List List_Create (char Name[]);
```

Crée et retourne une liste de type non ordonné, initialement vide. On pourra y insérer des objets en utilisant la procédure `List_Insert`. Le paramètre `Name` sert à identifier la liste lors des traces et des rapports. On recommande que le nombre de caractères ne dépasse pas 32.

```
List_List List_CreateOrd (List_OrdProc Precede, char Name[]);
```

Crée et retourne une liste de type ordonné, initialement vide, et dans laquelle on pourra insérer des objets en utilisant la procédure `List_InsertOrd`. Cette liste sera maintenue en ordre automatiquement, selon la fonction `Precede` fournie par l'usager. On peut passer directement l'identificateur de notre fonction comme argument en autant qu'elle soit compatible avec la définition du type `List_OrdProc`. Cette fonction doit retourner la valeur `1` si et seulement si l'objet `A` doit précéder l'objet `B` dans la liste. Pour le paramètre `Name`, voir `List_Create`.

```
void List_CollectStat (List_List L);
```

Déclenche un recueil automatique de statistiques pour la liste `L` (déjà créée). Lorsqu'on appelle cette procédure, deux blocs statistiques sont créés et initialisés. L'un (de type `Stat_Accumulate`) sert à mesurer l'évolution de la taille de la liste en fonction du temps, et l'autre (de type `Stat_Tally`) échantillonne les durées de séjour des objets dans la liste. Ce dernier ne recueille comme observations que les durées de séjour des objets insérés dans la liste *après* l'appel à `List_CollectStat`, et qui *quittent* la liste pendant la période d'observation, c'est-à-dire entre la dernière initialisation de ce bloc et l'instant courant. Les fonctions `List_StatSize` et `List_StatSojourn` retournent les blocs statistiques en question. `List_CollectStat` appelle aussi automatiquement `List_InitStat` pour initialiser ces deux blocs et faire une mise à jour pour le premier. Lorsqu'on appelle cette fonction, il est recommandé de le faire tout de suite après la création de la liste.

```
void List_InitStat (List_List L);
```

Réinitialise les deux blocs statistiques associés à la liste `L`, par des appels à `Stat_Init`, et fait une mise à jour pour celui qui mesure l'évolution de la longueur de la liste. Ces deux blocs doivent avoir été créés auparavant par un appel à `List_CollectStat`.

```
void List_Init (List_List L);
```

Vide la liste `L` de tous ses objets et réinitialise ses blocs statistiques (s'il y a lieu). Ne détruit pas, cependant, les objets qui s'y trouvent. Ces objets peuvent donc servir ultérieurement à d'autres fins. Seul l'usager peut détruire, lorsqu'il le désire et afin de récupérer l'espace qu'ils occupent, les objets qu'il a lui-même créés.

```
void List_Insert (void * P, List_Position Where, List_List L);
```

Insère l'objet `P` dans la liste `L`. Selon la valeur du paramètre `Where`, l'objet est inséré au début de la liste (`List_First`), à la fin (`List_Last`), avant l'objet courant (`List_Previous`) ou après l'objet courant (`List_Next`). L'objet `P` devient l'objet courant. La liste `L` doit être de type non ordonné, c'est-à-dire avoir été créée à l'aide de la procédure `List_Create`. Note: si `Current` est passé en paramètre la procédure ne fait rien.

```
void List_InsertOrd (void * P, List_List L);
```

Insère l'objet `P` dans la liste `L`. Cette liste doit être de type ordonné (créée par la procédure `List_CreateOrd`), et est donc maintenue en ordre selon la fonction d'ordonnancement fournie par l'usager.

`int List_Belongs (void * P, List_List L);`

Prend la valeur `1` si l'objet `P` est dans la liste `L`, sinon prend la valeur `0`. Lorsque la fonction prend la valeur `1`, `P` devient l'objet courant, sinon l'objet courant demeure inchangé.

`void * List_View (List_Position Which, List_List L);`

Permet d'observer et de retourner un objet qui se trouve dans la liste `L`. Selon la valeur du paramètre `Which`, cet objet sera le premier objet de la liste (`List_First`), le dernier (`List_Last`), l'objet courant (`List_Current`), celui qui précède l'objet courant (`List_Previous`) ou celui qui le suit (`List_Next`). La valeur `NULL` sera retournée si on tente d'observer l'objet précédent le premier de la liste, ou celui suivant le dernier de la liste, ou encore si on tente d'observer un objet dans une liste vide. Si l'objet retourné est différent de `NULL`, alors il devient l'objet courant.

`void * List_Remove (List_Position Where, List_List L);`

Retire et retourne un objet de la liste `L`. Selon la valeur du paramètre `Where`, cet objet sera le premier objet de la liste (`List_First`), le dernier (`List_Last`), l'objet courant (`List_Current`), celui qui précède l'objet courant (`List_Previous`) ou celui qui le suit (`List_Next`). La valeur `NULL` sera retournée si on tente de retirer l'objet précédent le premier de la liste, ou celui suivant le dernier de la liste, ou encore si on tente de retirer un objet dans une liste vide. Si l'objet retiré est l'objet courant et que la liste n'est pas vide, le nouvel objet courant sera le suivant de celui retiré (ou le dernier de la liste, si l'objet enlevé était le dernier).

`void List_RemoveObject (void * P, List_List L);`

Retire de la liste L un objet particulier `P`, spécifié par l'usager. Une erreur sera signalée si on tente de retirer un objet d'une liste vide, ou un objet ne faisant pas partie de la liste. Cette procédure regarde d'abord si l'objet courant est l'objet cherché, sinon elle parcourt la liste pour le retrouver. Lorsqu'on retire l'objet courant, le nouvel objet courant est choisi de la même façon que dans `List_Remove`.

`unsigned long List_Size (List_List L);`

Retourne le nombre d'objets se trouvant dans la liste L.

`void List_Concatenate (List_List * L1, List_List * L2);`

Concatène la liste `L2` à la fin de la liste `L1`, puis détruit `L2`. `L1` et `L2` doivent avoir été créées par `List_Create` (c'est-à-dire être non ordonnées). L'objet courant de `L1` n'est pas modifié. Si des statistiques étaient recueillies sur `L1`, elles sont réinitialisées.

`void List_Sort (List_List L, List_OrdProc Precede);`

Trie la liste `L` selon la fonction d'ordonnancement `Precede` fournie par l'usager. On peut passer directement l'identificateur de notre fonction comme argument en autant qu'elle soit compatible avec la définition du type `List_OrdProc`. Cette fonction doit retourner la valeur `1` si l'objet A doit précéder l'objet `B` dans la liste `L`. La liste `L` doit être de type ordonné (créée à l'aide de la procédure `List_CreateOrd`). La fonction `Precede` utilisée lors du tri peut être différente de celle spécifiée lors de la création. Dans ce cas, la liste est tout simplement retriée selon la nouvelle fonction, qui sera utilisée par la suite.

```
void List_Merge (List_List * L1, List_List * L2);
```

Fusionne les listes ordonnées `L1` et `L2` en une seule liste ordonnée `L1`, puis détruit `L2`. `L1` et `L2` doivent être toutes les deux ordonnées selon la même procédure `Precede` fournie par l'usager. La nouvelle liste s'appellera `L1` et sera triée selon l'ordre des listes de départ. L'objet courant de `L1` n'est pas modifié. Si des statistiques étaient recueillies sur `L1`, elles sont réinitialisées.

```
void List_Split (List_List *L, List_List *L1, List_List *L2,
                 char Name1[], char Name2[]);
```

Divise la liste `L` en deux nouvelles listes `L1` et `L2`, à partir de l'objet courant de `L`, puis détruit la liste `L`. Ces deux nouvelles listes auront le même type d'ordonnancement que la liste L. La liste `L2` contiendra l'objet courant de `L` et tous ses suivants, dans le même ordre, et la liste `L1` contiendra tous les objets précédant l'objet courant de `L`, dans le même ordre que dans `L`. Les objets courants de `L1` et `L2` deviennent leurs premiers objets respectifs. Les paramètres `Name1` et `Name2` servent à donner des descripteurs aux nouvelles listes, pour les identifier lors des traces à l'écran. Si des statistiques étaient recueillies sur `L`, elles sont réinitialisées et on en recueillera sur `L1` et `L2`.

```
void * List_FindFirstSuchThat (List_List L, List_CondProc Cond);
```

Parcourt la liste `L` linéairement à partir de l'objet courant (inclusivement), et appelle la procedure Cond (P) pour chacun des objets P de la liste et retourne le premier objet pointé par `P` dans la liste tel que `Cond (P) = 1`. Si aucun des objets à partir de l'objet courant jusqu'à la fin de la liste ne satisfait la condition, la valeur de l'adresse retournée est `NULL`. Si on veut que toute la liste soit parcourue, à partir du début, on peut appeler `List_View` juste avant l'appel de cette procédure. A noter que la procédure `Cond` ne doit pas modifier (par des effets de bord, par exemple) la valeur du pointeur sur l'objet courant de la liste `L`, sinon les résultats obtenus sont imprévisibles.

```
void List_ForEachInListDo (List_List L, List_ActionProc Action);
```

Parcourt la liste `L` linéairement à partir de l'objet courant (inclusivement), et appelle la procédure `Action (P)` pour chacun des objets `P` de la liste, à partir de l'objet courant jusqu'à la fin de la liste. Si on veut que toute la liste soit parcourue, à partir du début, on peut appeler `List_View` juste avant l'appel de cette procédure. A noter que la procédure `Action` ne doit pas modifier (par des effets de bord, par exemple) la valeur du pointeur sur l'objet courant de la liste `L`, sinon les résultats obtenus sont imprévisibles.

```
Stat_Block List_StatSize (List_List L);
```

Retourne le bloc statistique qui mesure l'évolution de la longueur de la liste en fonction du temps. Il s'agit d'un bloc de type `Stat_Accumulate`. Ce bloc n'existe que si on a déjà appelé `List_CollectStat` pour cette liste. Sinon, cette fonction imprime un message d'erreur.

```
Stat_Block List_StatSojourn (List_List L);
```

Retourne le bloc statistique qui mesure les durées de séjour des objets dans la liste `L`. Il s'agit d'un bloc de type `Stat_Tally`. Ce bloc n'existe que si on a déjà appelé `List_CollectStat` pour cette liste. Sinon, cette fonction imprime un message d'erreur.

`void List_Report (List_List L);`

Fournit un rapport statistique complet sur la liste `L`. On doit avoir appelé `List_CollectStat` auparavant pour cette liste.

`void List_Delete (List_List * L);`

Vide la liste `L` de tous ses objets et détruit cette liste. Tout comme `List_Init`, cette procédure ne détruit pas les objets se trouvant dans cette liste.

References

[1] J. H. Ahrens and U. Dieter. Computer methods for sampling from gamma, beta, poisson and bionomial distributions. *Computing*, 12:223–246, 1972.

[2] J. H. Ahrens and U. Dieter. Computer methods for sampling from the exponential and normal distributions. *Communications of the ACM*, 15:873–882, 1972.

[3] J. H. Ahrens and U. Dieter. Extension of Forsythe's method for random sampling from the normal distribution. *Math. Comput.*, 27(124):927–937, Oct. 1973.

[4] J. H. Ahrens and U. Dieter. Computer generation of poisson deviates from modified normal distributions. *ACM Trans. Math. Software*, 8:163–179, 1982.

[5] J. H. Ahrens and U. Dieter. Generating gamma variates by a modified rejection technique. *Communications of the ACM*, 25:47–54, 1982.

[6] R. W. Bailey. Polar generation of random variates with the t-distribution. *Mathematics of Computation*, 62(206):779–781, 1994.

[7] D. J. Best and N. I. Fisher. Efficient simulation of the von Mises distribution. *Appl. Statist.*, 28:152–157, 1979.

[8] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81:637–654, 1973.

[9] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29:610–611, 1958.

[10] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, second edition, 1987.

[11] B. W. Brown, J. Lovato, K. Russel, and J. Venier. *RANDLIB.c — Library of C Routines for Random Number Generation*. The University of Texas, M. D. Anderson Cancer Center, Houston, 1997. Version 1.3.

[12] P. Busswald. *Effiziente ZUfallszahlen-, erzeugung aus der Zetaverteilung und der, verallgemeinerten Poissonverteilung*. PhD thesis, Techn. Universitaet Graz, Graz, Austria, 1993. 152 pp.

[13] R. C. H. Cheng. The generation of gamma variables with non-integral shape parameter. *Appl. Statist.*, 26:71–75, 1977.

[14] R. C. H. Cheng. Generating beta variates with nonintegral shape parameters. *Communications of the ACM*, 21:317–322, 1978.

[15] J. Dagpunar. *Principles of Random Variate Generation*. Oxford University Press, 1988.

[16] J. S. Dagpunar. An easily implemented, generalized inverse gaussian generator. *Commun. Statist. Simul.*, 18(2):703–710, 1989.

[17] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.

[18] L. Devroye. Random variate generators for the poisson-poisson and related distributions. *Computational Statistics and Data Analysis*, 8:247–278, 1989.

[19] D. Duffie. *Dynamic Asset Pricing Theory*. Princeton University Press, second edition, 1996.

[20] J. M. Chambers et al. A method for simulating stable random variables. *J. Amer. Statist. Assoc.*, 71:340–344, 1976. Correction: J. Amer. Statist. Assoc. 82 (1987), 704.

[21] G. W. Hill. Algorithm 395: Student's t-distribution. *Communications of the ACM*, 13:617–619, 1970.

[22] N. L. Johnson. Systems of frequency curves generated by methods of translation. *Biometrika*, 36:146–176, 1949.

[23] V. Kachitvichyanukul and B. Schmeiser. Computer generation of hypergeometric random variates. *J. Statist. Comput. Simul.*, 22:127–145, 1985.

[24] V. Kachitvichyanukul and B. W. Schmeiser. Binomial random variate generation. *Communications of the ACM*, 31(2):216, February 1988.

[25] A. W. Kemp. Efficient generation of logarithmically distributed pseudo-random variables. *Applied Statistics*, 30:249–253, 1981.

[26] C. D. Kemp. A modal method for generating binomial variables. *Commun. Statistics, Theory and Methods*, 15:805–813, 1986.

[27] W. J. Kennedy Jr. and J. E. Gentle. *Statistical Computing*. Dekker, New York, 1980.

[28] A. J. Kinderman and J. F. Monahan. New methods for generating Student's-t and gamma variables. *Computing*, 25:369–377, 1980.

[29] R. Kremer. *C-Rand: Generatoren für nicht–gleichverteilte Zufallszahlen*. PhD thesis, Technical University Graz, Graz, 1989. 152 pp.

[30] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, second edition, 1991.

[31] P. L'Ecuyer. Combined multiple recursive random number generators. *Operations Research*, 44(5):816–822, 1996.

[32] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[33] P. L'Ecuyer and C. Lemieux. Variance reduction via lattice rules. *Management Science*, 46(9):1214–1235, 2000.

[34] P. L'Ecuyer and C. Lemieux. Variance reduction via lattice rules. *Management Science*, 46(9):1214–1235, 2000.

[35] K. Lehnen. *Erzeugen von Zufallszahlen fuer zwei exotische Verteilungen*. PhD thesis, Technical University Graz, Graz, Austria, 1989. German, 107 pp.

[36] C. Lemieux and P. L'Ecuyer. Randomized polynomial lattice rules for multivariate integration and simulation. submitted for publication, 2001.

[37] J. F. Monahan. An algorithm for generating chi random variables. *ACM Trans. on Math. Software*, 13:168–172, 1987.

[38] J. S. Ramberg and B. W. Schmeiser. An approximate method for generating asymmetric variables. *Communications of the ACM*, 17:78–82, 1974.

[39] H. Sakasegawa. Stratified rejection and squeeze method for generating beta random numbers. In *Ann. Inst. Statis. Math.*, volume 35 B, pages 291–302, 1983.

[40] J. L. Schonfelder. Chebyshev expansions for the error and related functions. *Mathematics of Computation*, 32:1232–1240, 1978.

[41] I. H. Sloan and S. Joe. *Lattice Methods for Multiple Integration*. Clarendon Press, Oxford, 1994.

[42] E. Stadlober. Sampling from poisson, binomial and hypergeometric distributions: Ratio of uniforms as a simple and fast alternative. Technical report, Math. Stat. Sektion, Forschungsgesellschaft Joanneum, Graz, 1989.

[43] E. Stadlober and R. Kremer. Sampling from discrete and continuous distributions with C-Rand. In *Simulation and Optimization, Springer Lecture Notes Econom. Math. Systems*, 1992.

[44] E. Stadlober and F. Niederl. Generation of non-uniform random variates with C-RAND. Technical Report Research Report No. 15, Institute of Statistics, Technical University Graz, Lessingtrasse 27, A-8010 Graz, Austria, July 1994. e-mail: fstat@ftug.dnet.tu-graz.ac.at.

[45] E. Stadlober and H. Zechner. Generating beta variates via patchwork rejection. *Computing*, 50:1–18, 1993.

[46] H. M. Taylor and S. Karlin. *An Introduction to Stochastic Modeling*. Academic Press, San Diego, third edition, 1998.

[47] H. Zechner. *Efficient Sampling from Continuous and Discrete Unimodal Distributions*. Technical, University Graz, Graz, Austria, 1994. 156 pp.